2009

WS-Pro: a Petri net based performance-driven service composition framework

Jinchun Xia
Iowa State University

WS-Pro: a Petri net based performance-driven service composition framework

by

Jinchun Xia

A dissertation submitted to the graduate faculty

in partial fulfillment of the requirements for the degree of

DOCTOR OF PHILOSOPHY

Major: Computer Science

Program of Study Committee:
Carl K. Chang, Major Professor
Robyn R. Lutz
Andrew S. Miner
Arun K. Somani
Dan Zhu

Iowa State University

Ames, Iowa

2009

DEDICATION

To my family.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

ACKNOWLEDGEMENTS

This thesis would not have been possible without the help and support from many people.

My utmost gratitude goes to my advisor, Dr. Carl K. Chang, who has been guiding me throughout my long journey in the software engineering field. He has been a wonderful questioner, great mentor, and excellent source of knowledge to me. I and other labmates greatly benefit from his questions and gradually learned the way to think in depth. He opened the academic world to me from many aspects. The comprehensive training I received from him is invaluable that will benefit my life. I also want to thank Dr. Chang for his care to students. He always listens to our problems and helps resolve our confusions. His everlasting enthusiasm and energy to his life and to his work encourage many of our student colleagues.

I am indebted to my committee members, especially Dr. Andrew S. Miner. This thesis greatly benefits from his detailed comments. The discussion with Dr. Miner is always constructive and inspiring. I learned from him the way to do research more rigorously, specifically, and thoroughly. I also enjoyed Dr. Miner's lectures that are of the best courses I ever took. My thanks also goes to Dr. Robyn R. Lutz. She is always kind and never hesitated to offer her help, even outside the classroom and the school. Her comments always hit the point. The discussion with her is so pleasant and efficient.

I wish to thank Dr. Mitra and all the students in our lab. Dr. Mitra is not only a faculty mentor, but also a good friend to me. He and other students in our lab create a friendly and fun working environment. That motivates me to stay in the lab longer than in my apartment. I will always remember my good time spent in the Software Engineering Lab.

I also extend my thanks to my friend, Xiaoyang Gu, who has given many important comments to this thesis. The friendship from Xiaoyang and many other people is one of the best

things I have ever owned in my life.

I am, and will always be grateful to my family – my husband Bo Liu, my parents, and my little brother – for their endless love and support. Without their support I would not be able to finish this work.

ABSTRACT

As an emerging area gaining prevalence in the industry, Web Services was established to satisfy the needs for better flexibility and higher reliability in web applications. However, due to the lack of reliable frameworks and difficulties in constructing versatile service composition platform, web developers encountered major obstacles in large-scale deployment of web services. Meanwhile, performance has been one of the major concerns and a largely unexplored area in Web Services research. There is high demand for researchers to conceive and develop feasible solutions to design, monitor, and deploy web service systems that can adapt to failures, especially performance failures. Though many techniques have been proposed to solve this problem, none of them offers a comprehensive solution to overcome the difficulties that challenge practitioners.

Central to the performance-engineering studies, performance analysis and performance adaptation are of paramount importance to the success of a software project. The industry learned through many hard lessons the significance of well-founded and well-executed performance engineering plans. An important fact is that it is too expensive to tackle performance evaluation, mostly through performance testing, after the software is developed. This is especially true in recent decades when software complexity has risen sharply. After the system is deployed, performance adaptation is essential to maintaining and improving software system reliability. Performance adaptation provides techniques to mitigate the consequence of performance failures and therefore is an important research issue. Performance adaptation is particularly meaningful for mission-critical software systems and software systems with inevitable frequent performance failures, such as Web Services.

This dissertation focuses on Web Services framework and proposes a performance-driven

service composition scheme, called WS-Pro, to support both performance analysis and performance adaptation. A formalism of transformation from WS-BPEL to Petri net is first defined to enable the analysis of system properties and facilitate quality prediction. A state-transition based proof is presented to show that the transformed Petri net model correctly simulates the behavior of the WS-BPEL process. The generated Petri net model was augmented using performance data supplied by both historical data and runtime data. Results of executing the Petri nets suggest that optimal composition plans can be achieved based on the proposed method.

The performance of service composition procedure is an important research issue which has not been sufficiently treated by researchers. However, such an issue is critical for dynamic service composition, where re-planning must be done in a timely manner. In order to improve the performance of service composition procedure and enhance performance adaptation, this dissertation presents an algorithm to remove loops in the reachability graphs so that a large portion of the computation time of service composition can be moved to a pre-processing unit; hence the response time is shortened during runtime. We also extended the WS-Pro to the ubiquitous computing area to improve fault-tolerance.

CHAPTER 1. INTRODUCTION

This chapter discusses research challenges in Service-Oriented Computing (SOC), introduces the motivation of this dissertation, and outlines the approach of this research.

1.1 Service-Oriented Computing

Reuse and integration have become two major concerns of software engineering in recent years. With the widespread use of software in modern society, software systems are becoming more and more complex and heterogeneous. It is usually difficult and financially inefficient for a sole organization to develop and maintain a complex software system for its own use; moreover, sometimes it is necessary to provide one-stop shopping for customers who require different functionalities from different organizations. Meanwhile, industry has been facing the challenge of rapid development in order to adapt to the fast changing business environment (for example, change of business model, shift of supplier, emergence of new competitor, etc). Rapid development usually demands fast requirement analysis and implementation based on off-the-shelf software.

Besides its importance in the business world, integration is a compelling internal concern of any organization due to the existence of legacy systems. Legacy systems are regarded as major bottlenecks of rapid software development. As reported by Forrester research on their survey conducted on the government and companies of all sizes (from small, medium, large to global), more than half of the companies surveyed have legacy systems running core business applications. Moreover, around 80% of their IT budget was spent on maintaining and upgrading legacy systems, leaving only 20% for new development (77)(78). Another survey conducted by CIO magazine suggested that 85% of companies have IT project backlog, and

they prefer integrating existing systems to buying new ones (39). The integration allows them to keep the intractable legacy systems running and to smoothly upgrade the old systems to newer business process and technologies.

Unfortunately, integration is an intricate and difficult issue. Software programs to be integrated can be completely heterogeneous. They may be implemented in various platforms, programming languages, communication protocols, interfaces, etc. Most integration needs to be done in the across-organization and across-location fashion. This type of corporation requires a mechanism to seamlessly integrate software applications together.

Responding to this demand, Service-Oriented Computing (SOC) has been proposed and researched in recent years. In SOC, services are basic building blocks for software development. Services are self-describing and self-containing software applications published in a repository by different providers. By wrapping software applications into services, implementation details in frameworks, languages, interfaces and others can be hidden. Services are functionally independent and loosely coupled. The software architecture built on the concept of service-orientation is called Service-Oriented Architecture (SOA). SOA greatly improves the integration process by bringing the integration process from IT level to the business level. Researchers and practitioners both regard SOA promising in promoting efficient software development (72).

Building on the great success of XML (11) and HTTP, Web Services have been accepted as the best way to implement SOA and have became the new trend for e-business. Web Services has a rich set of XML-based protocols and languages. Through a specific interface, called UDDI (Universal Description, Discovery and Integration) (102), software components are published as services. Using standard protocols, including SOAP (Simple Object Access Protocol) (97) and WS-BPEL (Web Services Business Process Execution Language) (81), etc., different services can cooperate or be composed to form a whole piece to fulfill greater needs.

In Web Services, integration is achieved through the process of service composition, which is the core activity in service-oriented computing. The key point in this process is automation. Researchers are trying to build a expressive and powerful framework so that services can be automatically located, understood, and seamlessly combined. Services should be able to

communicate freely and semantically unambiguously. Though great progress has been made in the past several years, there are still many issues requiring attention in Web Services research.

1.2 Performance Engineering in Web Services

Smith defined "performance" (96) as "the degree to which a system or component accomplishes its designated functions within given constraints, such as speed, accuracy, or memory usage". Increasing awareness of performance has been noticed in industry. As our dependency on software increases, so does the size and complexity of software systems. This completely makes it more and more difficult to develop, test, and maintain these software systems. The cost of software failure tends to be unacceptable and unpredictable. Srivastava and Koehler (99) described the demand from industry for robust and verifiable web services with high performance. Ludwig also pointed out that performance is "typically the first QoS aspect that needs to be addressed" (63). When it comes to the Web Services world, performance becomes an even more urgent issue because we observe more performance failure due to heterogeneous underlying frameworks.

Though with the promising advantages of integration and reuse, the Web Services framework introduces new challenges to our performance study. Unlike distributed software, Web Services have unique characteristics, such as loose coupling, statelessness, and dynamic composition, which makes it more intricate to analyze and guarantee service performance. In a distributed software system, developers have full control of all components. Thus the assurance of the service quality becomes a relatively easier internal task. Software systems can be thoroughly tested to ensure their correctness and their possession of certain properties. However, in a Web Services framework, component services are provided by different service providers which may spread around the world. The across-organization and across-location properties fail many existing performance analytic techniques in service-oriented computing.

Performance engineering in Web Services pertains to issues ranging from performance analysis, performance testing and performance monitoring to runtime adaptation. Because of dynamic service composition, most composite services are composed and delivered at runtime,

which diminishes the effectiveness of software testing. Until now, the most effective method to evaluate web services performance is runtime monitoring (5). With this circumstance, well-planned performance design and performance adaptation are critical to the success of Web Services. By integrating performance analysis into the service design, we hope to mitigate the risk of performance failure. At the same time, an effective performance adaptation mechanism aims at improving system tolerance on performance failure.

1.3 Research Questions and Challenges

We identify three major issues in performance engineering of web services. Performance modelling, performance-driven service composition, and performance adaptation.

1.3.1 Performance modelling

Traditionally, a common method used to evaluate system performance is performance testing, which can be only carried out after the software system is implemented. Performance requirements, as part of the core system requirements, tend to be overlooked during the design stage. When performance problems are disclosed during testing, software engineers have to face the high cost of redesign. To address this problem, many software researchers, motivated by the immaturity of research in software performance testing (105), devoted much effort to study software system performance analysis in the last decade. Unlike run-time performance testing, performance analysis, which relies on performance modelling, can be performed at early stages of software development to help developers evaluate software designs and avoid the high cost of redesign.

With performance modelling, we can predict the performance of a design to a certain extent and identify potential performance bottlenecks and design faults. With this information, we can refine our software design. Since this re-design happens before we put effort into implementation, the cost is relatively low. In this way, we reduce the cost of project development and mitigate the risk of software failure. A well founded and well executed performance analysis plan is very important to large-scale software development.

As a major mechanism to predict system performance, performance analysis provides important support for impact analysis. As reported by the Standish Group, 11.8% of the failure causes of failed projects can be attributed to changing requirements (100). Changing requirements cause the software system to either fail to deliver some functionalities or fail to meet quality criteria. Impact analysis techniques enable software engineers to evaluate the functional and non-functional impact of speculative changes on a software system. While functional impact is given primary attention, impact on system performance is often neglected. Introducing changes to the system without non-functional impact analysis can be extremely harmful because the changes may degrade the QoS level of the system or even cause system failure. Fortunately, with performance analysis, software engineers can predict performance of the system with speculative changes instead of real ones, evaluate system designs, and make decisions on alternative architectures, reducing the risk of project failure (19).

Though it shows great promise to promote software quality, performance modelling is an error-prone process that requires special expertise. This difficulty limits its use in industry. Instead, researchers have been seeking to derive analytic models from existing software architecture models (108) (107). In the Web Services paradigm, the business process is such a model to depict compositional architecture of aggregated services. Once written in WS-BPEL (81), a business process can describe how the component services are composed together and how they interact to achieve functional goals. Therefore, the first issue of performance engineering is to accurately and efficiently transform WS-BPEL into a performance modelling language. In other words, we intend to generate a performance model from the business process written in WS-BPEL. The specific performance model we are interested is Petri net (51) (76).

The correctness of the transformation is the key concern as the generated model provides the basis for analysis and computation. To the best of our knowledge, none of the existing works ever proved the correctness of the transformation, manual or automatic, from business process to analytical model.

1.3.2 Performance-based service composition

In the service composition procedure, a service provider holds a business process, and needs to bind each component task to a real service. After the binding is finished, an execution plan is formed. The binding process converts the business process to a concrete service. Speaking of performance engineering, we need to guide the composition procedure so that the execution plan has optimal performance.

The performance of the service composition procedure itself is another issue that also requires attention. In dynamic composition, which is one of the design goals of Web Services, the computation of the execution plan is actually included in the response time to the user. Therefore, the performance of our composition algorithm should be given equal priority with the overall performance issue.

1.3.3 Service adaptation

As discussed above, Web Services framework is a dynamic environment with heterogeneous platforms. Service failure, either functional failure or performance failure, is a norm. In order to provide continuous service to our customers with acceptable performance, we need adaptation mechanisms. The adaptation mechanism is usually called re-planning because we basically re-plan the service composition at runtime. Existing works do not provide a satisfactory solution for this problem. One reason is the poor performance of the re-planning procedure itself.

1.4 Research Goals

Our research goal is meant to venture into the many challenges to developing a comprehensive solution. This dissertation addresses the performance engineering issue of service composition to improve the overall performance of the whole web service framework.

We will first take the business process as an architecture model and transform it to a performance model. The performance model we use here is Petri net in favor of its modelling and analytic power. This transformation bridges software design and system property analysis. After generating Petri net, we can use it to verify whether the service composition is function-

ally correct, and whether it suffers deadlock (107). We can also use it to check the stochastic properties of the composite service and to identify performance bottlenecks.

The key point for this work is to ensure that our transformation is sound. In other words, the generated Petri net must correctly simulate system behavior described in the business process.

Though verifying the correctness of service composition is a big concern in the Web Services research, the verification activity itself does not depart from the model checking research that has been pursued for decades (85). We do not discuss how to used the generated Petri net for performance prediction and system property verification in this dissertation. Interested readers can refer to our previous publication (107) and other related works (35). In this dissertation we only focus on how to correctly generate Petri net model that maintains the fidelity of the business process behavior.

We also design algorithms to efficiently compute the optimal execution plan of the composite service, and provide timely adaptation to performance failure. Our approach can be summarized as follows.

1. Defining a formalism to transform WS-BPEL 2.0 to Petri net, which enables property verification. We also demonstrate the soundness of this transformation - we prove that the state-transition systems of the business process and Petri net are isomorphic. The transformation and proof are conducted in an inductive fashion.

2. Designing a two-phase algorithm to compute execution plan. This algorithm is based on Petri net model. We design an algorithm to remove loops in the state graph so that major computation can be moved to the offline phase. This algorithm ensures optimal runtime computation.

3. Designing a performance-based adaptation algorithm. This algorithm takes business alliances and financial cost into account.

4. Evaluating WS-Pro through experiments and simulation. Some small-scale projects were used to evaluate the performance of the two-phase composition algorithm. We simulated

large-scale service environment to evaluate the effectiveness of the adaptation algorithm. Both experiments generated promising results.

1.5 Dissertation Organization

The rest of this dissertation is organized as follows.

- Chapter 2: A research background introduction to Web Services, service composition, and Petri net.
- Chapter 3: A summary of representative work related to this research in the areas of performance modelling, business process modelling, and service composition/adaptation.
- Chapter 4: Detailed transformation from WS-BPEL to Petri net. A state-transition based proof serves to demonstrate the soundness of the transformation. The transformation is conducted based on constructs (basic activities and structured activities).
- Chapter 5: A two-phase composition algorithm that can compute the execution plan to achieve optimal performance is introduced. Preliminary evaluation results of this composition algorithm are reported in this chapter. We also present the adaptation algorithm and simulation results.
- Chapter 6: As an extension, the WS-Pro is applied in the ubiquitous computing area to help improving fault-resilience.
- Chapter 7: We conclude this research, list contributions and discuss several directions for future extension work.

CHAPTER 2. BACKGROUND OF RESEARCH

Web Services are the target system of this research and performance is the problem we try to address. This research heavily relies on the modelling language of Petri net. This chapter provides some background information of Web Services and Petri net.

2.1 Web Services

SOA proposes an important concept to build software in the service-oriented way. This concept can be implemented using a wide range of available technologies, including CORBA (20), RPC (93), RMI (26), Web Services (9) and DCOM (40). Among them, Web Services are usually the best option. Building on the http and XML, Web Services have a rich set of protocols covering service description, composition, publishing, communication, service level, security, governance, etc. These xml-based protocols includes WSDL (Web Service Description Language) (106) for service description, UDDI (Universal Description, Discovery and Integration) for service publishing and finding (102), SOAP (Simple Object Access Protocol) (97) for communication, and WS-BPEL (Web Services Business Process Execution Language Version 2.0) for business process description (81), etc. With these lingua francas, Web Services become the best way to implement SOA. They are strongly loose-coupled and address the problem of software integration.

In terms of software integration, most of the technologies available for building SOA are working at the IT level. Web Services have a complete stack of protocols from the business level to the lower communication level. With these protocols, Web Services bring the integration from the IT (Information Technology) level up to the business level, hence achieving better interoperability and higher flexibility. We mentioned the need for rapid development to adapt

in the fast changing business environment in chapter 1. Web Services are designed from the ground focusing on the integration issue. It allows service providers to easily tune their business model, replace technologies, and shift platforms, etc.

Another important feature of Web Services is loose coupling. In frameworks such as CORBA and RPC/RMI, developers have to generate server code (skeleton) and client code (stub) for a single method. All communication between service providers and service requesters has to go through the server code and client code. Service requester needs to obtain and deploy the stub in order to invoke the remote service. Alternatively, in Web Services, communication is through message passing. No pre-installation is needed. Generally speaking, CORBA is an Object-Oriented framework where software deals with objects. It allows integration independent of platform and language; however, this integration is achieved through tight coupling. Web Services are not centering around objects, but documents (such as SOAP message, WSDL description, etc.). With the same platform-independent and language-independent features, services are integrated through these documents (despite the abbreviation "SOAP" contains the word "object", Web Services do not deal with objects at all).

Software systems developed using traditional integration technologies are more focused on architecture design, just as we used to research on distributed software design patterns. With the evolution of these systems, architecture design is difficult and costly to modify. In Web Services, however, the issue of "architecture design" becomes less critical because it is easier to ship changes in business logic to the end product.

We have already discussed why Web Services are becoming the dominating software design paradigm. One may intuitively ask: "What is a web service?"

According to the definition by the w3 consortium (9), "A web service is a software system designed to support interoperable machine-to-machine interaction over a network. It has an interface described in a machine-processable format (specifically WSDL (106)). Other systems interact with the Web service in a manner prescribed by its description using SOAP (97) messages, typically conveyed using HTTP with an XML serialization in conjunction with other Web-related standards."

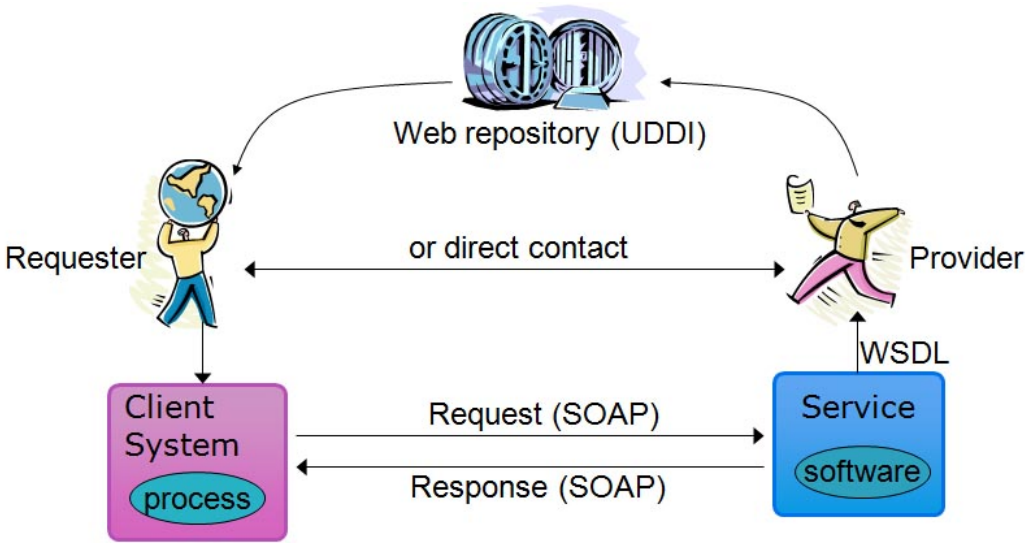


Figure 2.1 Web services scenario

As shown in Figure 2.1, in the framework of Web Services the service provider wraps its software as a service, describes the service using WSDL protocol, and then publishes it to the service repository called UDDI. The customer can search UDDI and find this service. Then the customer directly contacts the service provider requesting for the service. In this framework, the communication between different parties, such as requests and responses, are based on message passing. The messages have to follow a format called SOAP.

A service could be *basic* or *composite*. A basic service is a building block for a composite service. The process of integrating basic services is called service composition. A composite service can also be used as a basic service for other composite services.

2.1.1 Service composition

In Web Services, software integration is realized through the service composition process, which is the core activity in the Web Services framework. In this procedure, we have a business process defined using the WS-BPEL (81) language. A business process describes how the components services are composed together and how they interact to achieve functional goals. If we treat the composite service as the software product, then the business process serves as the model to describe the operational logic of the software system. In other words, it models

system behavior. With the business process defined, software developers need to bind each component task to a real service. After finishing binding, we have an execution plan. At this stage, the business process becomes a concrete service. We then complete the execution plan and deliver the service to the clients.

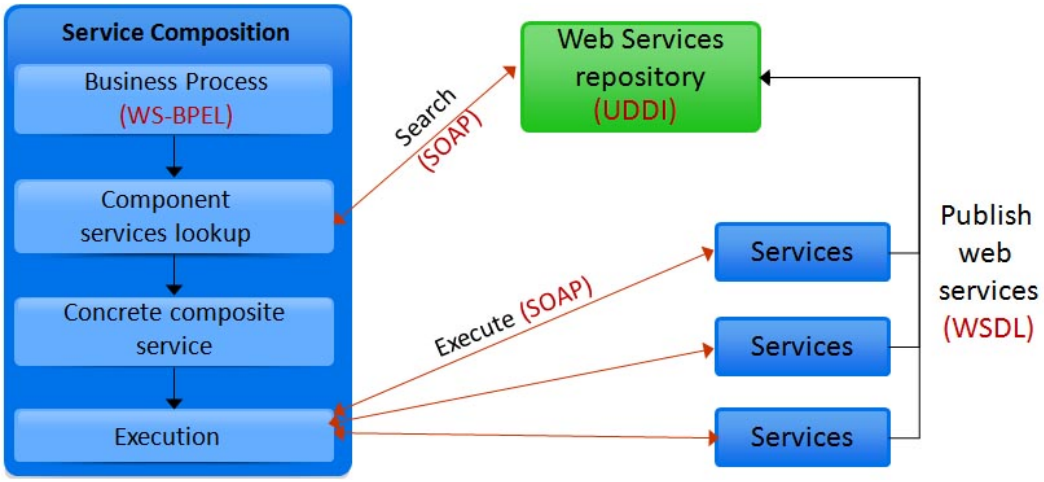


Figure 2.2 Services composition

The service composition raises many research problems including: how to discover available component services, how to select the ones best suitable for pre-defined goals, how to bind them, and how to adjust the composition when problems arising during execution. Ideally, this procedure will achieve "value-added" that is usually associated with the integration process.

The composition procedure can be manual or automatic. Currently most Web Services applications are based on manual composition. However, the ultimate objective and design goal of Web Services is automatic service composition, called *dynamic service composition*. Due to the gigantic amount of data over the internet, the number of available services can easily reach hundreds or thousands. For specialized applications, this number may be limited. However, the work of selecting component services and deploying the composite service is burdensome and error-prone. It is only feasible for small-size projects that have limited scope and corporation - usually with internal integration. The other major drawback is that manual composition cannot respond to runtime failure efficiently. Since manual composition is normally finalized at design time, it usually does not specify what actions should be taken when problems (service

failure, service overdue, etc.) arise during runtime execution.

Dynamic service composition aims at automating the complete procedure of integration including service discovery, service selection (with or without guidelines), service binding, service execution, and runtime adaptation.

Comparing these two types of composition, obviously the manual approach is much simpler and easier to handle, but lacks flexibility and scalability. Dynamic service composition provides the most desired features but adds complexity. Among all the issues necessary to realize dynamic composition, two issues gain the most attention and are critical to advance Web Services research:

- Dynamic service selection
- Runtime adaptation

Dynamic service selection includes two parts: selection of an execution path (if there are alternative ones), and selection of a provider for each component service in this path.

Runtime adaptation requires effective monitoring techniques, design of adaptation trigger, and adaptation mechanism. There might be varied considerations in the design of these issues. For example, if the criteria of adaptation is too strict we may interrupt the execution frequently, which may decrease the efficiency and waste resources. Contrarily, if the criteria is too loose, we may disappoint our customers. Another issue is whether we want the absolute optimal solution or near-optimal solution. Sometimes finding the absolute optimal solution takes long time may delay our response to the performance failure.

2.2 Petri net

As a mathematical and graphic tool, Petri net has been proved to be a very powerful modelling tool for a large variety of systems (50) (84) (85). Murata (76) described Petri nets as promising for modelling systems that are distributed, parallel, nondeterministic, or stochastic. Besides using them to model system behavior such as concurrency and asynchronies, Petri nets are also frequently used to model system properties such as performance (65). The graphic

representation allows Petri net to provide better pictorial view of architecture than the peer analytic model, Queueing Network, which only keeps the conceptual system architecture.

A Petri net can be formally defined as (84):

$PN = (P, T, I, O, M_0)$ where

–$P = \{p_1, p_2, \ p_n\}$, the set of places

–$T = \{t_1, t_2, \ t_m\}$, the set of transitions

–$I \subseteq (P \times T)$, the set of input arcs from places to transitions

–$O \subseteq (T \times P)$, the set of output arcs from transitions to places

–$M_0 = P \rightarrow \{0, 1, 2, \}$, the initial marking.

In Petri nets places can hold tokens. The distribution of tokens among all the places is called *marking* which describes the status of the system. When all the input arcs of a transition hold the required number of tokens, we say that this transition is enabled. An enabled transition can be fired, which moves tokens from the input places to the output places (Note the moving may not be equal. For example, the total number of tokens taken from the input arcs may be greater or less than the total number of tokens put into the output arcs).

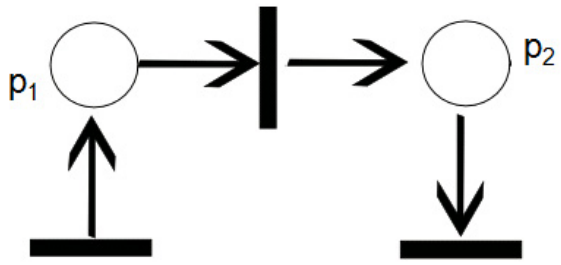


Figure 2.3 A Petri net that models a server.

In Figure 2.3, we have a Petri net that models a server. Whenever the lower left transition fires, one token is added to place p_1, which means one request comes in. One token in place p_1 means the server is busy (processing one task or multiple tasks). Therefore, we can interpret markings as follows:

- Marking *00* (both places are empty): the server is idling and the queue is empty.

- Marking *01* (p_1 is empty and p_2 holds one token): the server is busy and the queue is empty.
- Marking *10* (p_1 holds tokens and p_2 is empty): there are waiting requests in the queue and the server is idling.
- Marking *11* (p_1 is non-empty and p_2 holds one token): the server is busy and some other requests are waiting in the queue.

2.2.1 Generalized Stochastic Petri Nets (GSPN)

Various extensions to Petri net models have been proposed to address different modelling needs. One such enhancement is the Generalized Stochastic Petri Net (GSPN) which associates a random firing time to each transition. A GSPN is defined formally as a tuple (2):

$GSPN = (P, T, I, O, M_0, R)$ where

–P, T, I, O, M_0 are as in basic PN

–$R = r_1, r_2, , r_m$ is the set of firing rates associated with the transitions.

In GSPN transitions are classified into two groups: immediate and timed transitions. Immediate transitions have higher priority than timed transitions, fire in zero time and are labelled with a marking-dependent firing probability. Random, exponentially distributed firing times are associated with timed transitions. A marking in which immediate transitions are enabled is known as a vanishing marking, while a marking in which only timed transitions are enabled is known as a tangible marking. Vanishing states have sojourn times which are zero, while tangible states have nonzero exponentially distributed sojourn times. Using GSPN we can randomize arrival rates and processing delays.

A good property of GSPN is that its reachability graph can be turned into a Continuous-time Markov Chain (CTMC) by assigning to each edge a weight equal to the firing rate of the associated GSPN transition (2). CTMC is one of the most useful tools for numerical analysis. However, it is difficult to directly generate CTMC for middle and large scale systems. Compared to CTMC, the GSPN model is more straightforward to build. Therefore, the normal method is to create a GSPN model first, and then recognize its embedded CTMC.

Based on steady-state and transient-state distribution probabilities derived from CTMC, most performance analysis can be conducted.

CHAPTER 3. REVIEW OF LITERATURE

Business modelling and service composition must take into account performance adaptation. Our research aims to provide comprehensive performance solution that takes business process is presented in WS-BPEL, computes all the way to the final execution plan, and provides effective adaptation. No other work exists that is as comprehensive; however, there are many similar works with different aspects. By focusing on the three performance issues identified in chapter 1, we summarize the related work in the following three major categories: performance modelling, business process modelling, and service composition/adaptation.

3.1 Performance Modelling

Five major trends in performance modelling are studied in this section: Software Performance Engineering (SPE)-based, simulation model-based, Process Algebra-based, Queueing Network-based, and Petri net-based. We review their strengths and weaknesses. Some of the modelling methods reviewed in this section are created with brute-force way and some of them are generated from an architecture model (for the necessity of generating a performance model from an architecture model, see chapter 1). The architecture model considered here is the UML model that dominates architecture design in industry.

3.1.1 SPE-based approaches

As one of the earliest works in software performance engineering, SPE (96) is often quoted for its capabilities in modelling and analyzing system performance. Two major models, *System Execution Model* and *Execution Graph* are provided in SPE for system deployment and system behavior analysis. An Execution Graph follows the critical path of execution and offers

a best-case response time for a particular sequence. The System Execution Model considers queuing delay caused by resource contention and yields not only response time but also such metrics as utilization, throughput, and residence time. Both models offer fast analysis because of their computational simplicity. They are also organized around resource requirements so the effects of different hardware configurations can be studied easily. The drawback of SPE is limited support for scheduling algorithms, synchronous communication, random arrival and processing delay distributions. It is noted that neither of the two SPE models are able to predict system behavior under the overload condition. Moreover, neither of the two models provides the means to collect performance metrics that are essential to later performance evaluation. Therefore, in its original form, SPE does not provide adequate models to represent both performance properties and system architecture, nor does it supply the feedback mechanism. Further validation through extension work of SPE was deemed necessary and embarked later.

For example, inspired by SPE, Vittorio Cortellessa et al. introduced an incremental method, PRIMA-UML, to transform UML-represented architecture into SPE models (21) (23). Their approach contains two parts: UML2EG and UML2QN. UML2EG generates *Execution Graph*, which carries software architecture and behavior information, from use case and sequence diagrams. UML2QN generates Queue Networks, which contains hosts information from deployment diagrams. The two results are then integrated into the SPE process for evaluating system performance. Their approach bridges UML-based software architecture and existing performance models, thereby enabling performance evaluation of software architecture at design stage. The value of this approach is limited by the problems with SPE discussed above.

Besides SPE-based performance modelling, Vittorio Cortellessa et al. also created a framework (24) which is intended to encompass more ADLs and analytic models. Effort in this framework includes generating the SPIN (48) models from state machines and scenarios, and transforming *Amilia* textual description to Markov Chain models. Aiming at non-functional analysis, this framework propagates analysis feedback through different models to help software architecture evaluation. Central to this framework is the XML-based integration core, which

takes the XML representation of the architecture model and generates analytic models such as SPIN and Queueing Network models. SPIN model checker helps software engineers identify deadlocks and race conditions. They plan to integrate their PRIMA-UML, the SPE-based performance analysis approach, into this framework.

3.1.2 Simulation model-based approaches

Some researchers are devoted to generating simulation models directly from software architectural design. Among them, a formal simulation language called SimML (Simulation Modelling Language) (3) was combined with UML by Arief et al. to study the performance of a particular event sequence. In SimML, a sequence is built in a sequence diagram using SimML classifiers and operations. Each sequence simulation is capable of producing the average time it takes to process a job for a given arrival interval. SimML has certain advantages over SPE. It can evaluate different arrival distributions and processing profiles by employing random arrival time and process delay distribution through random variables. SimML does not assume balanced workload and is therefore able to calculate the average processing time even if conditions become overloaded. An integrated environment was created to support SimML design and simulation; however, there is no embedded metrics designed in the UML extensions of SimML, or a mechanism to feed the simulation results back to the software architecture design.

Unlike SPE, SimML does not consider resource requirements; instead it groups all delays into one variable, the processing delay. Therefore, in order to study the effects of alternative hardware, the analyst must guess at how the processing delay distribution will change, instead of just considering new resource characteristics. The lack of resource profiling also means that memory and disk usage requirements cannot be verified. SimML does not take into account various scheduling algorithms. It is not possible to investigate the behavior of synchronous communication because SimML is event based. Besides these disadvantages, SimML has its own design environment. A SimML model has to be manually created using its notations in order to run simulation.

Marzolla and Balsamo proposed a new process-oriented simulation model called UML-Ψ (70) (71). They developed a UML notation by modifying UML SPT (UML profile for schedulability, performance and time specification) (82). The simulation model is generated from annotated use case, deployment, activity, and collaboration diagrams. The transformation from the UML diagrams to the simulation model is automated by a tool. Simulation results are reported back and associated with UML diagrams. The authors proposed a new idea to validate the simulation models using the equivalence relations ≈M and ≈U. The ≈M relation judges whether two simulation models are equivalent and ≈U evaluates whether two UML diagrams are equal. They argued that the simulation models can be validated if two simulation models have the same structure and demonstrate equivalent performance, and their corresponding software architectures are equivalent as evaluated by ≈U. Unfortunately, the definitions of the equivalence relations were not shown or put into practice in their reported approach.

Another interesting study on extending UML to generate simulation models was reported in (73). The authors targeted real-time systems and defined stereotypes related to real-time domain constrains, such as period, deadline, jitter, etc. They used static diagrams, collaboration diagrams, classes, nodes, and associations to generate the simulation model. Scheduling policies are well presented in their work; however, the transformation from the extended UML model to the simulation model or analytic model was not specified in their paper. Since this approach was specifically designed for a real-time system, it is not suitable for studying general software systems.

3.1.3 Process Algebra-based approaches

Transforming software architecture into Process Algebra is another possible approach (7) (88) (13). Bennett and Field proposed an approach to transform UML sequence diagram into state machine upon which a finite state processes (FSP) (64) model is generated (7). The transformation, however, must be done manually.

Pooley (88) combined state diagrams with collaboration diagrams to generate a Process Algebra model for each combined diagram. The generated Process Algebra models are then

integrated together to create a single PEPA (Performance Evaluation Process Algebra) model. The difficulty of combining individual PEPA models was discussed in the approach. He also suggested a way to directly derive a continuous Markov Chain model from the combined state and collaboration diagrams.

The combination of UML 2.0 activity diagram and PEPA was studied in (13). The activity diagram underwent significant changes in UML 2.0 compared to its earlier representation in UML 1.x, thus presented a much higher level of complexity because it includes some high-level modelling techniques such as control flows and object flows. The authors specifically investigated the transformation from the UML 2.0 activity diagram to PEPA.

In Process Algebra-based approaches, usually stochastic behavior and resources are integrated into system architecture and the performance evaluation is based on the numerical calculation of the underlying Markov Chain. One major concern for this kind of approach is the state explosion problem well-known in the model checking area. In addition, the major drawbacks of Process Algebra-based approaches include a low degree of automation, lack of a feedback mechanism, and requirements of expert-level knowledge.

3.1.4 Queueing Network-based approaches

In order to harness the power and effectiveness of graph-analytic models such as Queueing Network (59) and Petri net (66), many researchers have tried to transform software architecture to these models and their variations. Among these models, Extended Queueing Network is gaining popularity (41) (86) (42) (89) (22) (52). By extending UML to represent performance properties, analytic models can be derived from UML diagrams and studied during the requirements, analysis, and design phases of the software lifecycle.

Pooley and King (89) suggested extending UML use cases for depicting workload, using sequence diagrams to trace simulation, and mapping state diagrams to Markov Chain models, etc. As an example, they derived a queueing network for the ATM machine example to illustrate their ideas. Though preliminary, these attempts sketched the possible transformation from UML to performance models.

Kahkipuro (52) (53) proposed a work to translate extended UML diagrams into AQN (augmented queueing networks), which represent simultaneous resource possessions. The augmented queueing networks are accepted by a decomposition algorithm in textual format to generate and solve queueing networks. Finally, the results are reported back in the sequence or collaboration diagrams. The transformation and the decomposition algorithm are supported by an automatic tool. However, this approach only supports textual representation, which degrades the quality of user interfaces. Furthermore, a set of systematic metrics and scheduling policies are needed for this approach to support comprehensive performance evaluation.

Some researchers consider software architecture patterns when evaluating their performance (41) (42). Dorina Petriu and his colleagues intended to generate Layered Queueing Network (LQN) from the annotated UML by using an intermediate format, Extensible Stylesheet Language Transformation (XSLT) (86). The analysis based on the LQN models can help evaluate performance of different software architecture patterns. This approach annotates UML collaboration, deployment and activity diagrams to present performance properties, while other diagrams such as use case and sequence diagrams, which illustrate important timing and behavior information, are not used. Moreover, this approach lacks the mechanisms to collect performance metrics or report performance results back to the architecture. The generated LQN has to be analyzed by an external tool.

Another study on evaluating software architecture patterns was reported in (38). Focusing on the component interconnection patterns, the authors annotate UML diagrams and map the annotated diagrams into extended Queue networks. Nevertheless, this study did not provide a systematic transformation from the annotated software architecture to performance models.

3.1.5 Petri net-based approaches

Unlike the conceptual graphical presentation of Queueing Network, Petri net maintains a better software architecture view. Interestingly, compared to the great amount of effort devoted to Queueing Network, less work has been done to transform software architecture into Petri net models. One common problem for existing work in this area is that most transformations are

based on UML statecharts, making the transformation process difficult because of the added complexity of statechart generation.

King and Pooley (54) (55) presented a work to derive Petri nets and a continuous Markov Chain using UML collaboration and statechart diagrams as the major source. The idea is to first embed the statecharts into collaboration diagrams to express the global state of a system. It then transforms the combined diagrams into GSPN models, which are finally united as a single GSPN model. The problem of this approach stems from identifying shared transitions to combine individual Petri nets together. Additionally, they did not provide a tool to automate the transformation from UML diagrams to Petri nets, nor an integrated environment to evaluate the generated Petri nets for performance predictions.

Lopez-Grao and his colleagues explored the possibility to formally transform different diagrams into LGSPN (Labeled GSPN) (61). The concept of this approach is to combine activity diagrams with statecharts to model possible execution paths. Statecharts are used as the high level model and activity diagrams are used for modelling internal flow process at lower level. In other words, an activity diagram describes a *doActivity* in the statechart. The activity diagrams may have hierarchy. After annotating performance requirements to the UML diagrams, each activity diagram is translated into a LGSPN. All the individual LGSPNs are finally combined and guided by the statecharts into a single performance model for the whole system or for a specific scenario. In this methodology, the transformation is automated by a CASE tool, ArgoUML. There is no support in this methodology to report performance results back into the software architecture. As pointed out by the authors, the LGSPN model has to be replicated for different *doActivity*/*subactivity* invocations. The inadequate expressing capability of LGSPN restricts the pattern of activities invocation (i.e., the invocation paths must be acyclic.)

Different from other approaches which focus on transformation from UML models, the approach created by Fukuzawa and Saeki (33) directly models the software architecture as Colored Petri net (CPN). The final model allows not only evaluation of software performance, but also other properties such as security and reliability. Nevertheless, their work seems pre-

liminary. No experiment has been conducted to verify whether this approach is practical.

There is one special technique which cannot be categorized into any of the five trends – the UML profile for schedulability, performance and time specification (UML SPT) (82) published by Object Management Group in 2002. UML SPT was published in response to the great demand and the fruitful research outcome of UML performance profile. It provides a way to annotate UML diagrams with performance requirements, system resources, and performance related behavior information. However, this profile only formalizes the way to extend UML diagrams, where no specific performance model can be created. Therefore, it lacks the capability to evaluate system performance. UML SPT was later integrated with some of the existing performance analysis techniques reviewed above (70) (71) (41) (86) (4) (7).

3.2 Modelling of the Business Process

Researchers have tried to model the business process using many analytical tools, such as abstract state machine (27), finite state machine (32), process algebra (30), and many more; however, most of them only worked on a subset of WS-BPEL language. For example, most of them do not consider control dependency, an important concept in WS-BPEL. Instead, we chose Petri net as our analytical model because of its powerful modelling capability and its nice graphic interface. Petri net has been proved to be very effective in stochastic modelling too. Milanovic and Malek summarized current web service composition solutions in (74). They compared Petri net with OWL-S, pi-calculus, and model checking/FSM, etc. They concluded that Petri net has good service connectivity, can prove composition correctness, but cannot specify non-functional properties, lacks the support of automatic composition, and has low degree of scalability. To the contrary of this statement, extended Petri nets such as stochastic Petri net and timed Petri net are well known for modelling stochastic properties of software system. Quantitative non-functional properties can be described and verified by an extended Petri net. As mentioned above, industry needs robust and verifiable web services with high performance (99). By using extended Petri nets, evaluation of web services performance and verification of service composition can be achieved at the same time. We summarize the

application of Petri net to service composition modelling below.

In fact, researchers have already tried to apply Petri net theory to web service composition. Martens defined usability, equivalence, and compatibility, etc., in (68). He used Petri net-based semantics to formally verify those properties. He and his colleagues also tried to translate BPEL4WS into Petri net(67). They used the "communication graph" defined in (68) to model the communication process among different services. This "communication graph" pairs communication links between service requesters and providers. However, the authors did not prove that the "communication graph" preserves the same behavior of the original BPEL process.

As an early experiment to model Web Services business processes in Petri net, a semantic model was proposed by Hamadi and Benatallah (46). This approach works directly on the composition of brute-force formed Petri net without the transformation from WS-BPEL. The authors of (46) modelled major business constructs but neglected many other semantics such as handler (compensation handler, fault handler, etc.) and control dependencies (control links). The constructs researched in (46) do not match the ones specified in WS-BPEL standards.

Research results reported in (94) and (104) (83) are most similar to ours. Both groups tried to transform WS4BPEL to Petri net. WS4BPEL is an obsolete version of WS-BPEL. Schmidt and Stahl (94) claimed that the generated Petri nets can be reduced in state space so that certain model checking tools can be used to verify system properties. They also created a tool under the open-source license, called BPEL2oWFN (its predecessor is BPEL2PN). However, Chitrakar reported that this tool generated incorrect Petri net Markup Language results (PNML) (8) that could not be accepted by other analytic tools (17). Coincidentally, Verbeek and Aalst (104) proposed the transformation from WS-BPEL to WF-net (103). As pointed out by Chitrakar, this approach introduced "skip-paths" which created a large number of unnecessary places and transitions (17). The result is Petri nets with big sizes in terms of places and transitions. Running computation on the big Petri nets is inefficient.

Lohmann and his colleagues improved BPEL2oWFN and expanded the model to adopt WS-BPEL 2.0 specification that was released in April 2007 (62). The Petri nets generated

from this work can have different patterns depending on the level of abstraction. The authors claimed that this approach is more effective in space reduction than the work reported in (104) and (83).

Different from the above approaches that focused on the control flow of WS-BPEL, Moser's work presented in (75) tried to model the data flow of WS-BPEL. They identified the data dependencies in WS-BPEL and represent them using CSSA (Concurrent Single Static Assignment Form) (60). The data dependencies were integrated into the mapping from WS-BPEL to Petri net. The authors argued that by including data dependencies their approach can avoid false-positive analysis results.

Some other researchers focused on using a formal model to predict service performance. The mathematical model presented in (92) was based on operations research techniques. It generates resources metrics such as utilization and throughput. Marzolla and Mirandola proposed in (69) another approach that applied Queueing Network techniques for performance prediction. In this approach, each service was modelled as a queueing network connected using edges according to the WS-BPEL description. Some experiments were conducted to evaluate the effectiveness of this approach. Koizumi and Oyama presented a similarly work using Timed Petri net (57). It is difficult to assess this work in terms of transformation from WS-BPEL to Timed Petri net because it is not documented with sufficient detail. This approach can simulate the generated Petri nets at different abstraction levels and generate response time according to different statistical confidence levels.

It is worth pointing out that almost none of the existing works considers the correctness of the transformation. The analytical model provides the basis for our verification and computation. It is crucial to ensure its correctness, which requires the soundness of the transformation process. In our approach, we use a state-transition based proof to demonstrate the soundness of transformation. This soundness distinguishes our approach from others. Additionally, we believe that Petri nets can be used to support the computation of the execution plan but as of this writing, not much work in the area has been reported.

3.3 Service Composition and Adaptation

Zeng and his colleagues divided service composition planning into two categories: local planning and global planning (116)(117). Local planning is limited to the task level at which the service composer computes QoS value of each candidate service and selects the best candidate for each specific task. Global planning takes the whole execution map into account and only computes QoS values for critical tasks. Integer Programming is used for global planning, however, the architecture of this approach is very simple and the authors did not make concrete design of the trigger for re-planning. It is worth pointing out that authors investigated the performance of the service composition procedure. Whereas poor performance was reported in this paper – re-planning may take longer duration than current execution.

The broker-based architecture proposed by Yu and Lin (115) targets composing, selecting, and adapting service composition. The service composition problem was modeled as a MCKP (Multiple Choice Knapsack Problem) and Pisinger's algorithm was used to solve it. Different from other approaches, this methodology takes the level of service into account when selecting services. This approach is designed for a multimedia domain so it includes some domain specific attributes in the QoS model. This approach did not consider re-planning.

Gokhale and Lu (37) modelled user group patterns using Customer Behavior Model Graph (CBMG) and transferring a CBMG into a discrete Time Markov Chain (DTMC) . The calculation of DTMC can reveal session time and availability based on pre-defined state availability parameters. This work is useful for understanding the impact of user pattern on system performance. However, this work only studies traditional e-commerce sites at high levels and does not fit in the Web Services framework.

Nguyen and his colleagues (80) proposed a disCSP-based methodology to solve the QoS-based service selection problem. Nested services and corporation among services are considered in this approach. They used disCSP to model the QoS problem and introduced Fuzzy theory to represent soft constraints on agents. A framework was developed to support this approach. Specifically focusing on re-planning, Canfora and his colleague (14) defined triggers and selection algorithms using Genetic Algorithm.

As discussed above, comprehensive performance management should include a selection program and a performance adaptation mechanism. The common problem with existing approaches is the neglection of the performance of the selection and adaptation procedures.

CHAPTER 4. TRANSFORMATION OF WS-BPEL TO PETRI-NET

The Web Services Business Process Execution Language (WS-BPEL) (81) is an executable language to describe business processes. It allows designers to write both executable and abstract business processes. WS-BPEL focuses on orchestration in the view of a central service and models the operational logic of a composed system, including when to invoke a service, whether to execute a service, or skip it, etc. This language supports programming structures such as IF and WHILE, and allows expression of parallel behavior (e.g., flow) and synchronization (e.g., control links). The syntax of WS-BPEL follows XML format (11). Below is an incomplete representation of this language. More details can be found in Appendix A. For complete syntax and semantics of WS-BPEL, interested readers can refer to (81).

WS-BPEL *::= U(O(import), O(documentation), O(partnerlink), O(variables), O(activities), O(handler), O(extension), O(messageExchanges), O(correlationSets))*

Activities *::= P(basic-activities | structured-activities)*

Basic-activities *::= Invoke | Receive | Reply | Assign | Throw | Wait | Empty | Extension activity | Exit | Rethrow*

Structured-activities *::= Sequence | If | While | RepeatUntil | Pick | Flow | ForEach | Compensate | CompensateScope*

standard-attributes *::= U(O(name), O(suppressJoinFailure))*

standard-elements *::= U(O(targets), O(sources))*

Invoke *::= "<invoke" invoke-attributes ">" invoke-elements "</invoke>"*

invoke-attributes *::= U(standard-attributes, partnerLink, O(portType), operation,*

O(inputVariable), O(outputVariable))

invoke-elements *::= U(standard-elements, O(correlations), O(catch), O(compensationHandler), O(toParts), O(fromParts))*

Where

O(x) ::= empty | x

#(x) ::= any number of x

P(x) ::= x #(x)

U(x,y) ::= any order of x and y

All the basic activities can be in the "<activity attributes />" form when there is no elements included. For example, the invoke activity may be defined as: "<invoke" invoke-attributes "/>".

A WS-BPEL process can be treated as an inductively defined system composed by multiple constructs (or called "activities") (79). In this section, we define the transformation from WS-BPEL to Petri net. In order to show that the generated Petri nets conform to the original system behavior, we demonstrate that the Petri net and the WS-BPEL process have isomorphic state-transition systems.

By induction, if we can prove the transformation of basic constructs is correct and this correctness remains during the inductive step, then we prove the soundness for transformation of the whole WS-BPEL process. The correctness of the transformation on constructs is discussed in the following of this section. The correctness of the inductive procedure is discussed in chapter 4.6.2.

4.1 Assumptions and Definitions

To ensure valid execution of a WS-BPEL process, we make the following assumptions.

Assumption 1. *The suppressJoinFailure is set to be "yes" so that bpel:joinFailure is not thrown when join condition is not satisfied; instead, the activity is skipped and the control passes on.*

Assumption 2. *The partnerLinks and correlation sets are always satisfied.*

Assumption 1 ensures that the business process will not be stopped due to join failure of links. Assumption 2 eliminates meaningless system states. If we model the mismatch of correlations, we will have a large number of system states that represent the skipping of activities due to mismatch of instances. This kind of skipping is meaningless to system analysis and adds complexity to our model.

Assumption 3. *All the basic services studied in this thesis must end.*

Assumption 4. *All the Petri nets generated in this thesis except PN_{exit} have a* start *place and an* end *place.*

As stated in the previous section, we aim to generate Petri net from WS-BPEL to model business process behavior. It is widely accepted that UML statecharts (58) effectively depict software behavior and system state transition. Meanwhile, reachability graph illustrates state transition in the Petri net world. Therefore, the problem of proving the correctness of transformation from WS-BPEL to Petri net becomes proving isomorphism of the statechart and the reachability graph.

WS-BPEL has concurrent activities that may include synchronization links. This complex situation is modeled in statechart with high level of abstraction, hence cannot illustrate the state transition with adequate detail. Moreover, reachability graphs of Petri nets model the state transition systems in the way of a state machine. A state machine can only be in one of the possible states at the same time, while the statechart can be in multiple states concurrently. Therefore, in this thesis, statecharts are used to model sequential state transition and state machines are used when concurrent state transition is involved (*Flow* and parallel *ForEach* activities). They are compared to the reachability graphs of the generated Petri nets. The isomorphism between the statechart / state machine and the reachability graph indicates the soundness of the transformation from WS-BPEL to Petri net.

Definition 1. *For a WS-BPEL construct P, its statechart is a 4-tuple SC_p = (S, A, S_0, S_e), where*

- $S = \{s_i \mid s_i$ is the i^{th} state, $0 \leqslant i \leqslant n\}$

- $A = \{a_{ij} \mid (a_{ij}$ *is the transition between* s_i *and* $s_j) \wedge (s_i, s_j \in S)\}$

- $S_0 = \{s_0\}$

- $S_e = \{s_n\}$.

The state transition system ST_{SC} *corresponding to this statechart is the 3-tuple* $ST_{SC_P} = (S, A, S_0)$*, where* S*,* A *and* S_0 *are defined as above.*

Definition 2. *For a WS-BPEL construct* P*, its generated Petri net is* PN_P*, where* $PN_P = (P, T, IN, OUT, R, M_0)$ *(*PN_P *will be defined according to each WS-BPEL construct) with reachability graph*

$RG_p = (M, EG, M_0)$

- $M = \{m_i \mid m_i$ *is the* i^{th} *marking of* PN_P*,* $0 \leq i \leq k\}$

- $EG = \{g_{ij} \mid (g_{ij}$ *is the transition between* m_i *and* $m_j) \wedge (m_i, m_j \in M)\}$;

- $M_0 = \{m_0\}$.

The state transition system corresponding to this Petri net is defined as:

$ST_{PN_P} = (M, EG, M_0)$

- *M, EG and* M_0 *are defined as above.*

Definition 3. *A graph* $G1 = (V_1, E_1)$ *is isomorphic, denoted as* $\cong$*, to another graph* $S = (V_2, E_2)$ *iff* $\exists$ *a bijection function* $f : V_1 \rightarrow V_2$ *such that*

for any $u, v \in V_1$*,* $(u, v) \in E_1$ *if and only if* $(f(u), f(v)) \in E_2$.

Theorem 1. ***(correctness)****: For a given WS-BPEL construct P, let* PN_P *be its Petri net presentation and* SC_P *be its statechart.* PN_P *simulates* SC_P *if* $ST_{PN_P} \cong ST_{SC_P}$*, where* $\cong$ *stands for isomorphic.*

Proof. Suppose $ST_{PN_P} \cong ST_{SC_P}$, based on the complete proof in (36), isomorphism implies bisimulation equivalence, which is denoted by $\sim$.

$ST_{PN_P} \cong ST_{SC_P}$

$\Rightarrow ST_{PN_P} \sim ST_{SC_P}$

$\Rightarrow PN_P$ simulates SC_P

□

Note that we do not need to prove the number of iterations for loop activities equal to the one in Petri net. Our purpose of transformation is to model the behavior, or the control flow, of different activities. In the Petri nets, we give different probability rates for staying or exiting the loop. These domain-specific probability rates will be used for the service composition at later stages.

There are two types of transitions in the Petri nets used in this thesis: timed transitions and immediate transitions. Timed transitions are illustrated in hollow rectangles, as shown in Figure 4.1. They are associated with firing rates, which are used to decide the probabilities of different branches.

4.2 Operational Semantics of WS-BPEL Activities

We borrow graphic items from flowcharts to visualize the control flow of WS-BPEL. Experienced readers may question whether the control flow depicts the correct semantics (someone may argue that the same WS-BPEL process can carry different semantics in different environments). To clarify this point, we can use Abstract State Machine (ASM) (44) to describe the operational semantics of WS-BPEL and illustrate the semantics in a control flow. Gurevich and Huggins has defined the semantics for C language using ASM (43). All the logic constructs defined in that work - for example, basic activities of invoke and receive, conditional expressions, logic branches and loops - are applicable for WS-BPEL. Farahdbod et al have formalized the operational semantics of BPEL using ASM (28) (29). They extended the original macro concept defined by Gurevich in (44) and described the state transition of BPEL control flow. Each BPEL activity is defined to have three possible states: enabled, disabled and completed. Fahland and Reisig refined this work by including events handling, dead-path elimination and correlation handling with a focus on "faulty" (27). We believe these works are enough to convey the operational semantics of WS-BPEL in control flows. Therefore, in the following we directly use the state transition models drawn in statecharts or state machines.

In the rest sections, we show how we transform basic activities and structured activities into Petri net. We also demonstrate the isomorphism between the statecharts of the original

activities and the reachability graphs of the generated Petri net. According to Theorem 1, this isomorphism suggests the soundness of the transformation.

4.3 Control links

4.3.1 Semantics of control links

Control dependency is an important feature of WS-BPEL. It is used to realize synchronization among parallel executing activities. Therefore, control links only pertain to "Flow" activities that have parallel execution paths. Each link has a source activity and a target activity.

Control links have the following characteristics.

(1) Synchronization. If a link has a source activity "A" and a target activity "B", then "B" has to wait till "A" finishes and passes link status (either *true* or *false*) to start its own execution.

(2) Cross boundaries. In WS-BPEL, control links may cross activity boundaries. For example, a flow with three *Sequence* activities can have control links between different execution paths[2]. These links are said to be crossing the boundaries of the *Sequence* activities.

Due to the cross-boundary property of control links, in our hierarchical transformation process, it is normal that we do not have complete link information when we transform a certain activity. For example, link "XtoY" directs from activity X to activity Y while X and Y locate in different *Sequence*'s. When we generate the Petri net for X, Y has not been transformed yet. In this case, it is hard to connect PN_x with PN_y instantly. Therefore, during transformation, we create Petri nets for all the component activities first regardless control links, and then add control links.

Control links do not cross the boundaries of loops or parallel executions. For example, we have a $While$ activity that has one component activity A. WS-BPEL semantics do not allow control links from outside $While$ activity to be associated with A. The outside links can only be associated with the $While$ activity itself. Similarly, WS-BPEL does not allow control links

[2]In WS-BPEL there should be no control link going backward in one single sequential path (81).

from outside a *Flow* activity to be associated with any component activity inside the *Flow* scope.

(3) Join conditions and transition conditions. Each target activity has a *join condition* that evaluates whether this activity should be executed or not. The *join condition* is expressed in WS-BPEL using the $< joincondition >$ element. Only when all the incoming links are *true*, the $< joincondition >$ is evaluated to true. If at least one control link has $false$ status, the $< joincondition >$ is evaluated to $false$. The $< joincondition >$ will not be evaluated unless all the incoming links have their status set. For example, activity A has n incoming links. At a certain moment, all the links have status $false$ except link l_n that still has *undecided* status. At this moment, even though we know that the $< joincondition >$ of A will eventually be evaluated to $false$, we wait for link l_n to gain its status.

On the source activity side, each outgoing link can be associated with a $< transitioncondition >$. When $< transitioncondition >$ is specified, the outgoing link is set to be *true* only when $< transitioncondition >$ is evaluated to true.

Note that control links leaving the same source activity may carry different link status depending on their corresponding $< transitioncondition >$.

4.3.2 Modeling control links in Petri net

The activity P with two incoming links and one outgoing link is modeled in Petri net as shown in Figure 4.1. In this Petri net, places *start*, $r1$, and *end* represent the three states of the activity P: start, running, and end. The place $r1$ may represent a sub-Petri net if the activity P is a structured activity.

4.3.2.1 Modeling incoming links

In Figure 4.1, the join condition of the activity P is represented in the two types of transitions *jf* and *jt*. Two incoming links form four possible input values for a join condition. Only one of the four inputs leads to a *true* join condition (when both incoming links are *true*). The other three inputs lead to a $false$ join condition. Therefore, in Figure 4.1, we have one *jt*

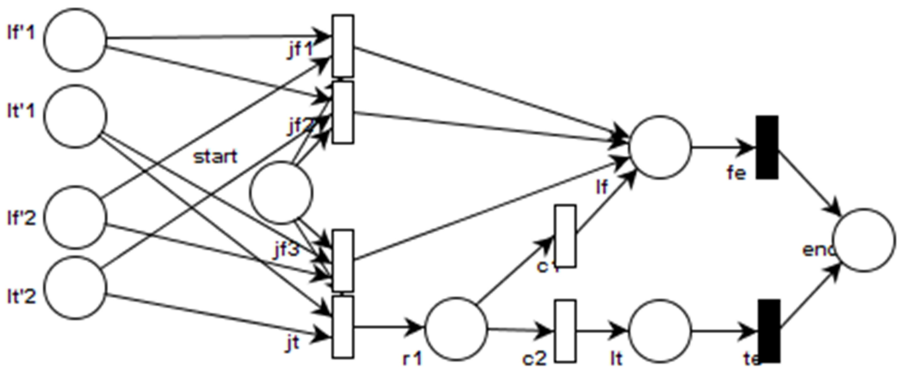


Figure 4.1 Modelling incoming control link status by Petri net.

transition corresponding to the input that leads to a *true* join condition, and three $jf_k|_{1\leq k\leq 3}$ transitions corresponding to the three inputs that lead to a $false$ join condition.

Whenever the join condition is evaluated to true, the jt transition fires. Otherwise jf_k transition is fired. The firing of jf_k skips the execution of this activity.

Places $lf'1$ and $lt'1$ in Figure 4.1 represent the status of incoming link 1. Similarly, places $lf'2$ and $lt'2$ represent the status of incoming link 2. These four places are connected to transitions jt and jf_k so that the following two requirements are met:

(1) Only when both incoming links have their status set (with a status other than "unset"), the join condition can be decided.

(2) Only when both incoming links have "*true*" status, jt is fired; otherwise one transition $jf_k|_{1\leq k\leq 3}$ is fired.

Each join condition transition is connected to all the incoming links ($lf'i$ or $lt'i$). Therefore, if one incoming link i has *unset* status, neither $lf'i$ or $lt'i$ holds a token. Consequently, none of the transitions is enabled. This way, Requirement (1) is satisfied.

The connection of $lf'i$, $lt'i$ to jt and jf_k ensures the satisfaction of requirement (2).

In a general case.

For l number of incoming links, there are totally 2^l possible link status, among which only one satisfies *true* link condition and all the rest lead to $false$ link condition. Therefore, in the Petri net, we have one jt transition and (2^l-1) $jf_k|_{1\leq k\leq 2^l-1}$ transitions.

The connection from $lf'i$ and $lt'i$ to jt and $jf_k|_{1\leq k\leq 2^l}$ is described in Algorithm 5.

4.3.2.2 Modeling outgoing links

After the execution of the activity, a *false* or *true* status is assigned to the outgoing link. We use two places, *lf* (link false) and *lt* (link true) to model link status. The *transition condition* of the outgoing link is modeled through Petri net transitions $c_k|_{1\leq k\leq 2}$. The enabling of transition *c1* means the *transition condition* is evaluated to false. The firing of *c1* will set the link to be *false* - putting a token in the lf place. Similarly, the firing of transition *c2* assigns value *true* to the outgoing link.

When *transition condition* is absent, the outgoing link is always set to *true*. In this case, the sub-net shown in Figure 4.2(a) is replaced by Figure 4.2(b).

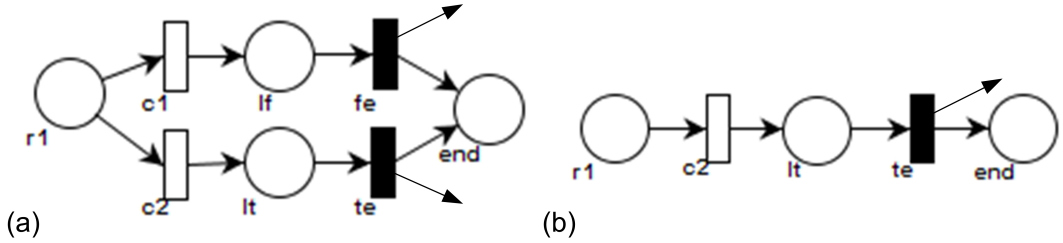


Figure 4.2 Modelling outgoing control link status by Petri net.

When both transitions *c1* and *c2* are enabled at the same time (place *r1* has one token), their firing rates will be used to decide which one to fire. The firing rates, which in this case are the different probabilities of "false transition condition" and "true transition condition" can be specified using domain-specific information or operational profile.

In a general case.

An activity with one outgoing link has two places indicating the status of the link: *lf* and *lt*. With multiple links, the two places are duplicated to illustrate status of different links. We have

$$N_{sp} = 2l \tag{4.1}$$

$$N_{st} = 2^l \tag{4.2}$$

where N_{sp} is the number of places representing link status, N_{st} is the number of transitions representing *transition conditions*, and l is the number of outgoing links.

Disclaimer:

Each WS-BPEL activity may have different incoming links and outgoing links. In our transformation, the modelling of activities without links is simpler to the one with links. The transformation and proof for activities with and without links are highly similar. For this reason, we only demonstrate linked activities in this thesis.

In the following, we describe atomic transformation for each type of WS-BPEL activity, and then elaborate how they are composed together. The composition follows *concatenation* and *insertion* rules as described in section 4.5.

4.4 Atomic Transformation

4.4.1 Basic activities

As mentioned in the previous section, we intend to model the behavior of a WS-BPEL process, i.e., its control flow. All the basic activities share the same syntax pattern except *invoke*, which represents the operation of invoking a remote service. *Invoke* has non-standard control elements such as *compensationHandler* and *catch*. The *compensationHandler* describes how to compensate a failed service and *catch* captures exceptional events. Both of them transfer control from the main activity to handlers described using other activities. However, as described in (81), an *invoke* activity can be transformed into a scope activity. Therefore, we can still express all the basic activities using the same syntax format and perform the same transformation.

All the basic activities may be associated with control dependencies. In the following of this section we examine *Basic Activities With Links* (*BAWL*).

4.4.1.1 Transformation

The syntax for basic activities with links (both incoming links and outgoing links) can be written as follows.

Basic-activities ::= O(Attrs) O(Elements) ACTivity

where $O(x)$ ::= *empty*|x; *Attris* denote attributes and *Elements* denote elements such as targets/sources and correlations. The corresponding operational semantics can be written using IF statement:

If *not(join condition)*

then *set linkFalse*

else *ACTivity*

If *not(transition condition)*

then *set linkFalse*

else *set linkTrue*

end If

end If

With this semantics, we can draw the control flow as in Figure 4.3. The Petri net representing the above semantics is shown in Figure 4.3(b). Its plain Petri net view is shown in Figure 4.3(c).

The definition of this Petri net is $PN_{BAWL} = (P, T, IN, OUT, R, M_0)$, where

P = *(start, r1, lt, lf, end)*

T = {*jt, jf, c1, c2, te, fe*}

IN = {*(start, jt), (start, jf), (r1, c1), (r1, c2), (lt, te), (lf, fe)*}

OUT ={*(jt, r1), (jf, lf), (c1, lf), (c2, lt), (fe, end), (te, end)*}

R = {r_{jt}, r_{jf}, r_{c1}, r_{c2}} *(firing rates for timed transitions)*

M_0 = {*[start]*}

All the labels of places and transitions here are for illustration purposes and will be changed accordingly when composed together into one complete Petri net for the whole WS-BPEL process.

In the Petri net shown in Figure 4.3, timed transitions are used to model control branches, such as $c1$ and $c2$. If the control flow is sequential (i.e., no competitive branches), an immediate transition is used, for example, *te* and *tf*.

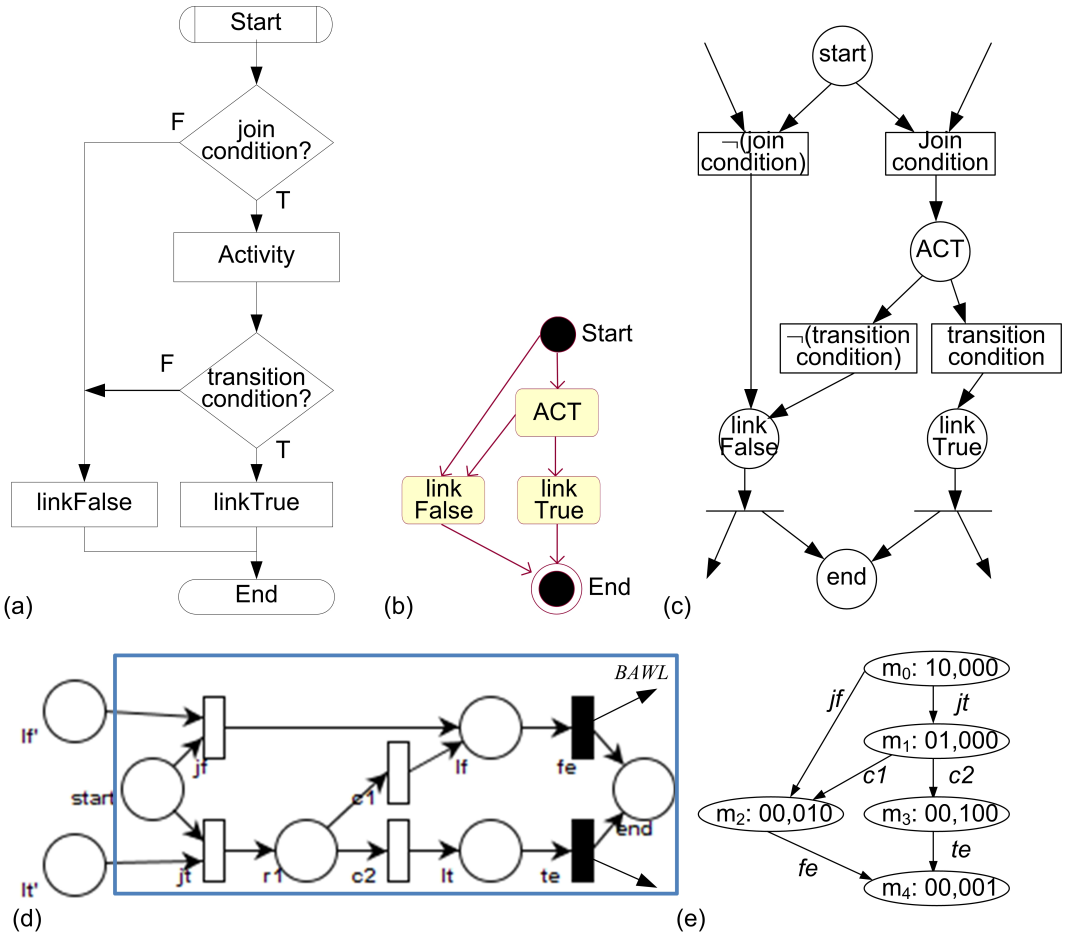


Figure 4.3 *BAWL* atomic activity. (a) control flow; (b) statechart SC_{BAWL}; (c) illustrating Petri net PN_{BAWL}; (d) Petri net PN_{BAWL}; (e) reachability graph RG_{BAWL}.

4.4.1.2 Correctness

Now we prove that PN_{BAWL} simulates the BAWL construct, i.e., the transformation is correct.

State Transition Systems. The transition systems $ST_{SC_{BAWL}}$ and $ST_{PN_{BAWL}}$ are shown as in Figure 4.4. Obviously, $ST_{SC_{BAWL}} \cong ST_{PN_{BAWL}}$ according to definition 4.1.

The proof of the transformation from WS-BPEL to Petri net is similar for most WS-BPEL constructs. Therefore, we will skip proofs in the rest of this thesis. Only statecharts and reachability graphs will be provided for illustration.

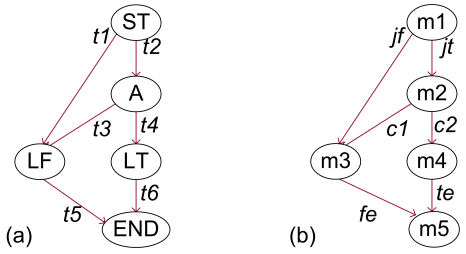


Figure 4.4 (a) $ST_{SC_{BAWL}}$; (b) $ST_{PN_{BAWL}}$.

4.4.1.3 Basic activities without links (BANL)

Now we study basic activities without links, *BANL*. Its control flow and Petri net representation are depicted in Figure 4.5.

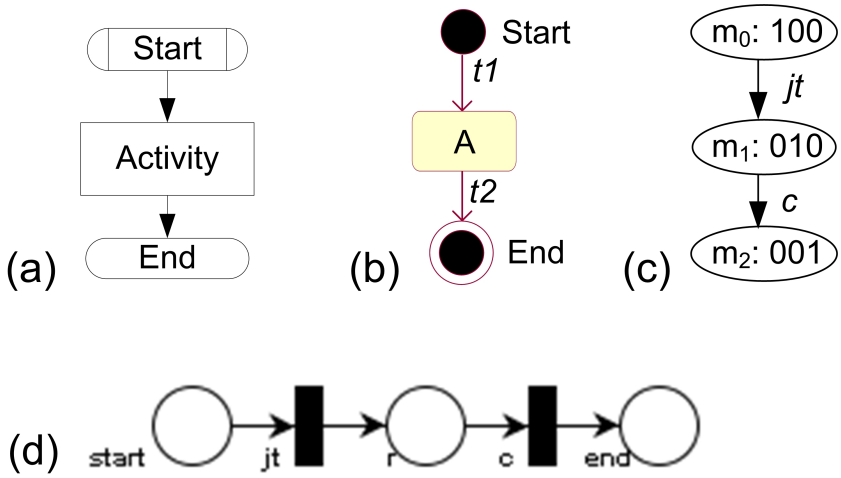


Figure 4.5 *BANL* atomic activities. (a) control flow; (b) statechart; (c) reachability graph; (d) Petri net.

The Petri net in 3(b) can be described as $PN_{BANL} = (P, T, IN, OUT, R, M_0)$, where

$P = \{start, r, end\}$

$T = \{jt, c\}$

$IN = \{(start, jt), (r, c)\}$

$OUT = \{(jt, r), (c, end)\}$

$R = \emptyset$

$M_0 = \{start\}$

4.4.1.4 Exit

The exit atomic activity is modeled as in Figure 4.6.

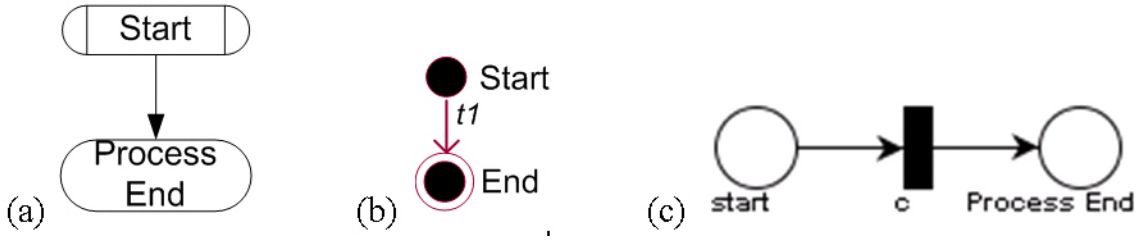


Figure 4.6 *Exit* atomic activities. (a) control flow; (b) statechart; (c) Petri net.

The Petri net in 4(b) can be described as $PN_{exit} = (P, T, IN, OUT, R, M_0)$, where

$P = \{start\}$(note that the "*process end*" place is the ending place for the whole service instance which is outside this Petri net)

$T = \{c\}$

$IN = \{(start, c)\}$

$OUT = \{(c, ProcessEnd)\}$

$R = \emptyset$

$M_0 = \{start\}$

In the rest of this section, we describe the transformation for structured activities. From the above analysis on basic activities we can see that an activity without links is just a simpler case of one with links. The transformation and proof for both cases are highly similar. For this reason, below we only show the linked version for structured activities.

Some structured activities share the same Petri net models. For example: $PN_{Sequence} = PN_{BAWL}$, $PN_{Pick} = PN_{if}$, $PN_{sequentialforeach} = PN_{while}$, and $PN_{parallelforeach} = PN_{flow}$ when *completioncondition* is absent. Though not shown in details in this thesis, we model handlers (exception handler, compensation handler, etc.) as branches after the main activity (similar to the *Scope* construct).

4.4.2 If

The syntax of the If construct is

If ::= "<if" standard-attributes ">"standard-elements condition Activities #(elseif) O(else) "</if>"

which is re-written as:

If *not(join condition)*

then *set linkFalse*

else If *(condition 1)*

activity 1

elseif *(condition 2)*

activity 2

else

activity 3

end If

If *not(transition condition)*

then *set linkFalse*

else *set linkTrue*

end If

end If

The *If* construct is modeled as in Figure 4.7. According to the WS-BPEL definition of construct *If*, there may be any number of *elseif*'s. Here we only model one *elseif*. However, the transformation scales up with the number of *elseif* and the proof remains the same. The original flow chart is depicted in the left graph of Figure 4.7(a). Since we only analyze control flow, branches (1) to (4) can be simplified to three branches, as shown in the right graph of Figure 4.7(a). Both figures are equivalent in terms of control. The transformation is based on the simplified graph.

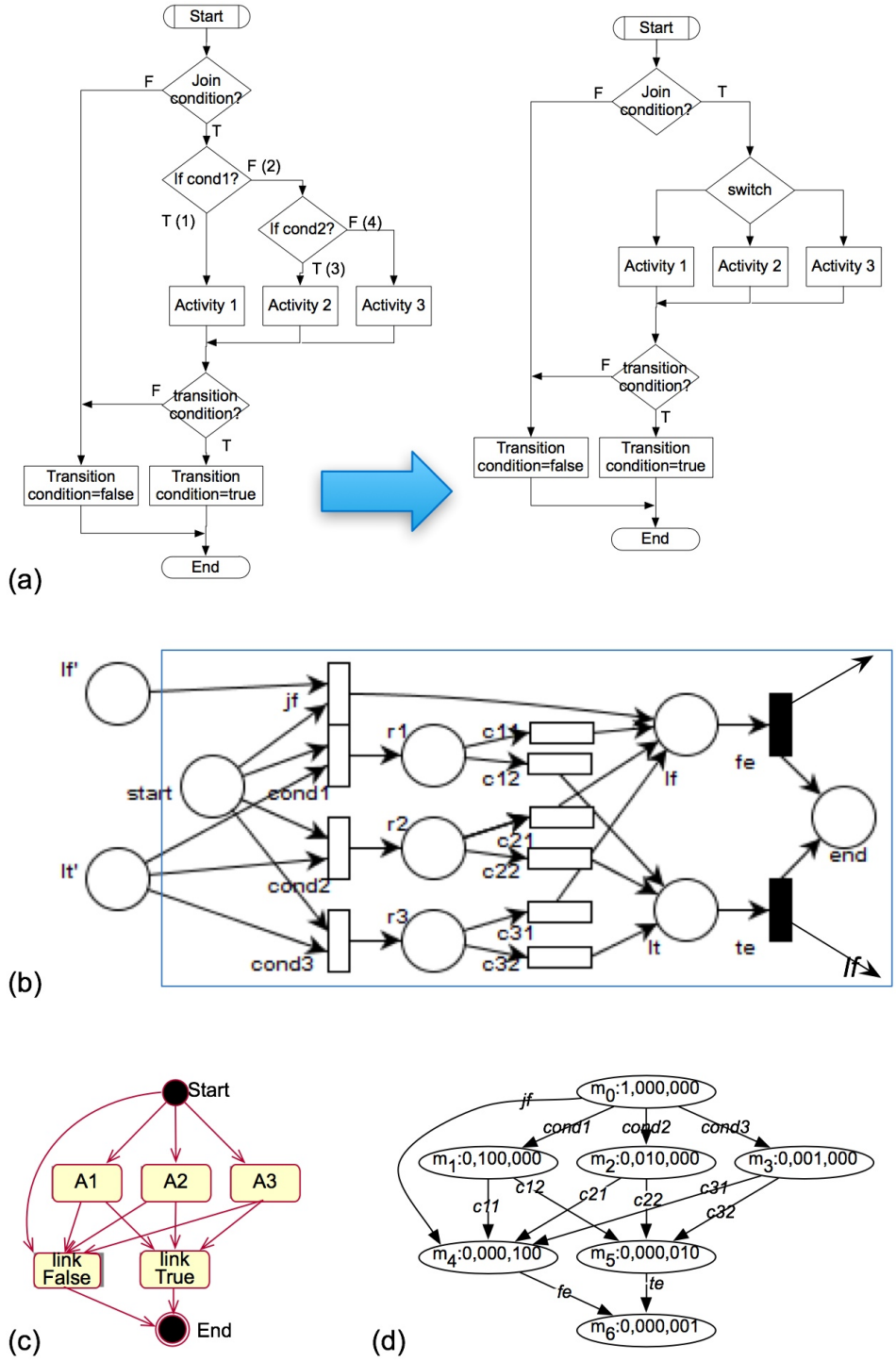


Figure 4.7 *If* construct. (a) control flow; (b) Petri net; (c) statechart; (d) reachability graph.

WS-BPEL does not prevent different branches from having the same condition. As a result, there might be race condition if multiple branches specify the same condition (Similarly, *Pick* activity may have race condition too). The handling of race condition is implementation dependent. WS-BPEL does not mandate any specific mechanism. In practice, branches with same condition are used to represent alternative execution paths.

PN_{if} = *(P, T, IN, OUT, R,* M_0*), where*

$P = \{start, \{r_i\}, lf, lt, end\}$

$T = \{\{cond_i\}, \{c_{i1}\}, \{c_{i2}\}, jf, jt, fe, te\}$

$IN = \{(start, jf), (start, \{cond_i\}), \{(r_i, c_{i1})\}, \{(r_i, c_{i2})\}, (lf, fe), (lt, te)\}$

$OUT = \{(jf, lf), \{(cond_i, r_i)\}, (\{c_{i1}\}, lf), (\{c_{i2}\}, lt), (fe, end), (te, end)\}$

$R = \{ r_{jf}, \{r_{condi}\}, \{r_{c_{i1}}\}, \{r_{c_{i2}}\}\}$ *(firing rates for timed transitions)*

$M_0 = \{[start]\}$

– $1 \leq i \leq n$, *where n is the number of "elseif" and "else" branches.*

– $\{r_i\}$ *is the set of places for "elseif" and "else".*

– $\{cond_i\}$ *is the set of transitions to start "elseif" and "else" branches.*

– $\{c_{i1}\}$ *and* $\{c_{i2}\}$ *are the sets of transitions to complete "elseif" and "else" branches.*

Proof skipped.

4.4.3 While

The syntax of the While construct is

While ::= "<while" standard-attributes ">" standard-elements condition Activities "</while>"

which is re-written as:

If *not(join condition)*

then *set linkFalse*

else *while (expression) do*

activities

endwhile

If *not(transition condition)*

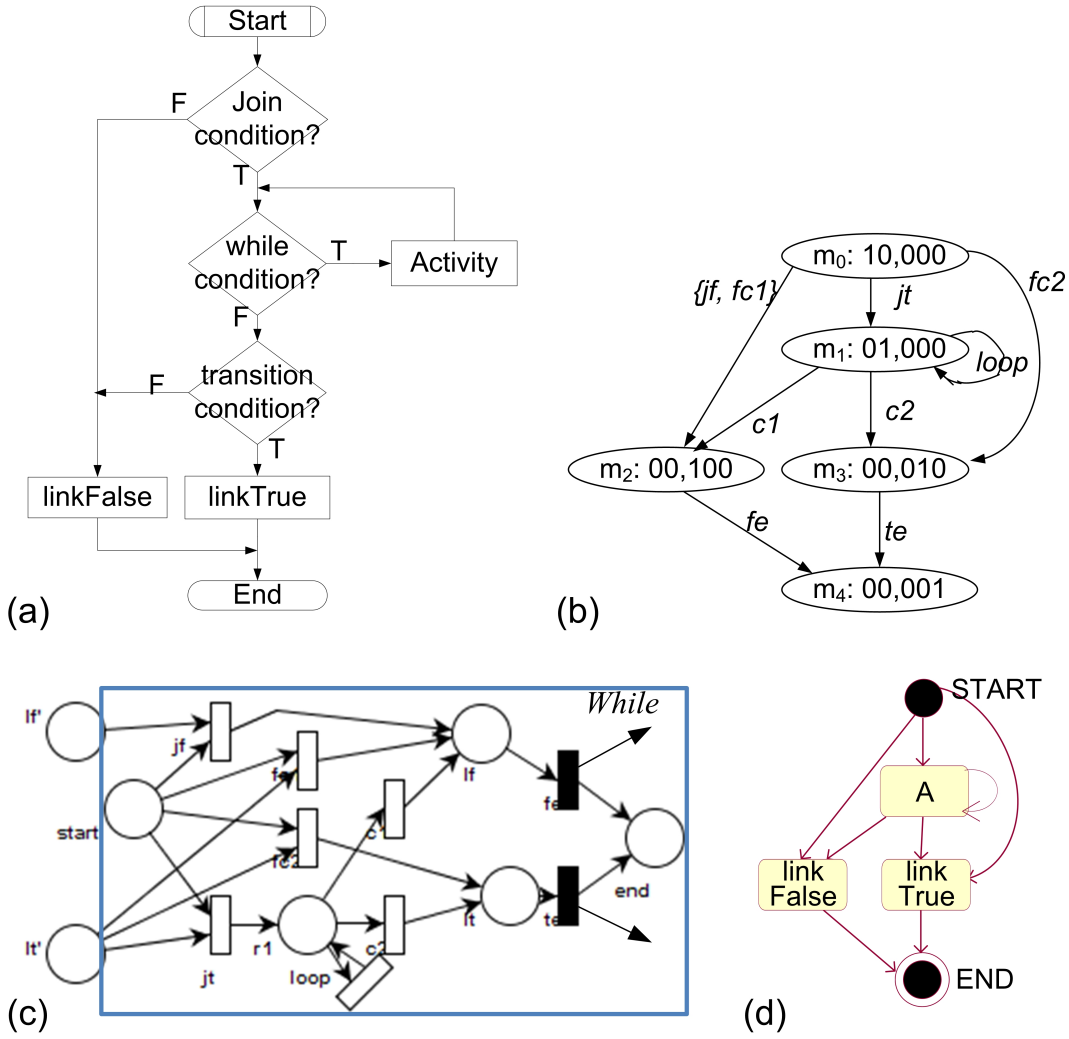


Figure 4.8 *While* construct. (a) control flow; (b) reachability graph; (c) Petri net; (d) statechart.

then *set linkFalse*

else *set linkTrue*

end If

end If

The *While* construct is modeled in Figure 4.8. We have:

$PN_{while} = (P, T, IN, OUT, R, M_0)$, where

$P = \{start,\ r1,\ lf,\ lt,\ end\}$

$T = \{jf,\ jt,\ fc1,\ fc2,\ loop,\ c1,\ c2,\ fe,\ te\}$

IN = {*(start, jf), (start, jt), (start, fc1), (start, fc2), (r1, loop), (r1, c1), (r1, c2), (lf, fe), (lt, te)*}

OUT ={*(jf, lf), (jt, r1), (fc1, fe), (fc2, te), (loop, r1), (c1, lf), (c2, lt), (fe, end), (te, end)*}

$R = \{ r_{jf}, r_{jt}, r_{fc1}, r_{fc2}, r_{loop}, r_{c1}, r_{c2}\}$

M_0 = {*[start]*}

Proof skipped.

The transitions *jf* and *jt* model the *join condition* just as other activities.

In the *While* construct, the *while condition* may be evaluated to false the first time. As a result, the activity may never be executed. Note that the *transition condition* is evaluated independent of the *while condition*. Even though the *while condition* is evaluated to false and the activity is never executed, the *transition condition* may still be evaluated to true. We use transitions *fc1* and *fc2* to model the case that initial *while condition* is *false* and the activity is skipped. The firing of *fc1* skips the execution of the activity and sets the outgoing link to be *false*. Similarly the firing of *fc2* skips the activity and sets the outgoing link to be *true*. *When the "While" activity does not contain any outgoing links, the* $fc1$ *and* $fc2$ *transitions are absent in* PN_{while}.

If the activity is being executed (place *r1* holds a token), three transitions are enabled, including *loop*, *c1* and *c2*. Transition *loop* revisits the loop. Transitions *c1* and *c2* terminate the iterative execution. Which transition to fire is decided by their firing rates that are set using domain-specific information or operational profile (from historical usage data).

4.4.4 RepeatUntil

The syntax of the RepeatUntil construct is

RepeatUntil ::= "<RepeatUntil" standard-attributes ">" standard-elements Activities condition "</RepeatUntil>"

which is re-written as:

If *not(join condition)*

then *set linkFalse*

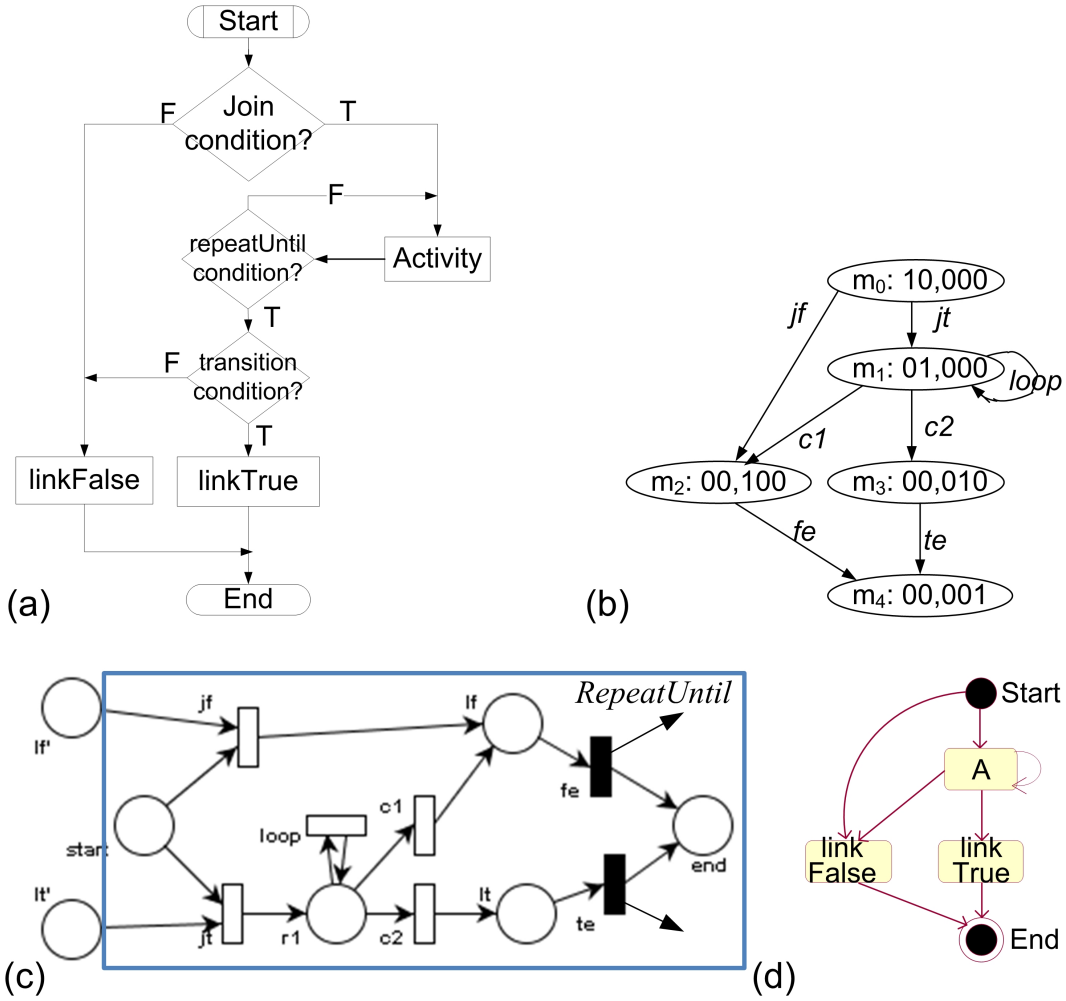


Figure 4.9 *RepeatUntil* construct. (a) control flow; (b) reachability graph; (c) Petri net; (d) statechart.

else *do*

activities

Untill (expression)

If *not(transition condition)*

then *set linkFalse*

else *set linkTrue*

end If

end If

The *RepeatUntil* construct is modeled in Figure 4.9. Note that the difference between PN_{While} and $PN_{RepeatUntil}$ is that PN_{While} has two extra transitions $fc1$ and $fc2$ that represent skipping the activity right from the beginning. In the *RepeatUntil* construct, the activity will be executed at least once. Therefore, we do not need the $fc1$ and $fc2$ transitions.

$PN_{RepeatUntil} = (P, T, IN, OUT, R, M_0)$, where

$P = \{start,\ r1,\ lf,\ lt,\ end\}$

$T = \{jf,\ jt,\ loop,\ c1,\ c2,\ fe,\ te\}$

$IN = \{(start,\ jf),\ (start,\ jt),\ (r1,\ loop),\ (r1,\ c1),\ (r1,\ c2),\ (lf,\ fe),\ (lt,\ te)\}$

$OUT = \{(jf,\ lf),\ (jt,\ r1),\ (loop,\ r1),\ (c1,\ lf),\ (c2,\ lt),\ (fe,\ end),\ (te,\ end)\}$

$R = \{\ r_{jf}, r_{jt}, r_{loop}, r_{c1}, r_{c2}\}$

$M_0 = \{[start]\}$

Proof skipped.

4.4.5 Pick

The syntax of Pick is:

Pick ::= "<pick" pick-attributes ">"U(standard-elements, P(onMessage), O(onAlarm)) "</pick>"

Each branch in the *Pick* activity is an event, either message or alarm. These branches may form a race condition. Similar to the If activity, the handling of the race condition is left to real implementation, hence not specified in the WS-BPEL.

We use a three-branch Pick construct as an example. Its control flow is depicted in Figure 4.10. This control flow has the same operational semantics with the *If* construct. Therefore, $PN_{Pick} = PN_{if}$, where

– $1 \leq i \leq n$, *where n is the number of "onMessage" and "onAlarm" branches.*

– $\{r_i\}$ *is the set of places for "onMessage" and "onAlarm" branches.*

– $\{con_i\}$ *is the set of transitions to start branches.*

– $\{c_{i1}\}$ *and* $\{c_{i2}\}$ *are the sets of transitions to complete branches.*

Proof skipped.

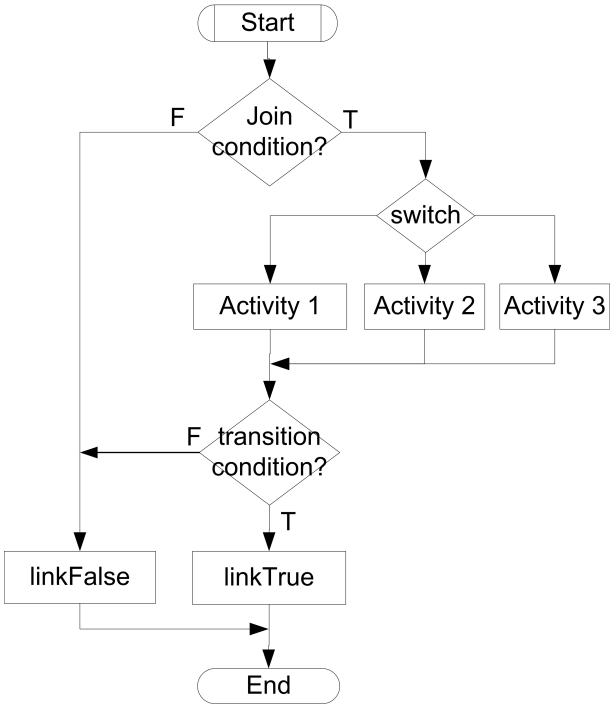


Figure 4.10 Control flow of *Pick* construct.

4.4.6 Flow

We use a three-parallel-branch *Flow* construct as an example, which is modelled in Figure 4.11. The syntax of flow is:

flow ::= "<flow" standard-attributes ">" U(standard-elements, O(links), Activities) "<flow>"

We use three places to represent the three execution paths, but in fact, we can model the parallel execution using only one place. Either way, we get the same state transition system. *Flow* is a structured activity that is composed hierarchically by other activities. Section 4.5 describes how a *Flow* activity is composed – each r_i place is replaced by a component Petri net. For this reason, we create one place for one execution path in PN_{flow} at the beginning so that we can insert the component Petri nets into PN_{flow} correctly.

$PN_{flow} = (P, T, IN, OUT, R, M_0)$, where

$P = \{start,\ \{r_i\},\ lf,\ lt,\ end\}$

$T = \{jf,\ jt,\ c1,\ c2,\ fe,\ te\}$

$IN = \{(start,\ jf),\ (start,\ jt),\ (\{r_i\},\ c1),\ (\{r_i\},\ c2),\ (lf,\ fe),\ (lt,\ te)\}$

$OUT = \{(jf, lf), (jt, \{r_i\}), (c1, lf), (c2, lt), (fe, end), (te, end)\}$

$R = \{r_{jf}, r_{jt}, r_{c1}, r_{c2}\}$ *(firing rates for timed transitions)*

$M_0 = \{[start]\}$

– $1 \leq i \leq n$, *where* n *is the number of parallel branches.*

– $\{r_i\}$ *is the set of places for parallel branches.*

Proof skipped.

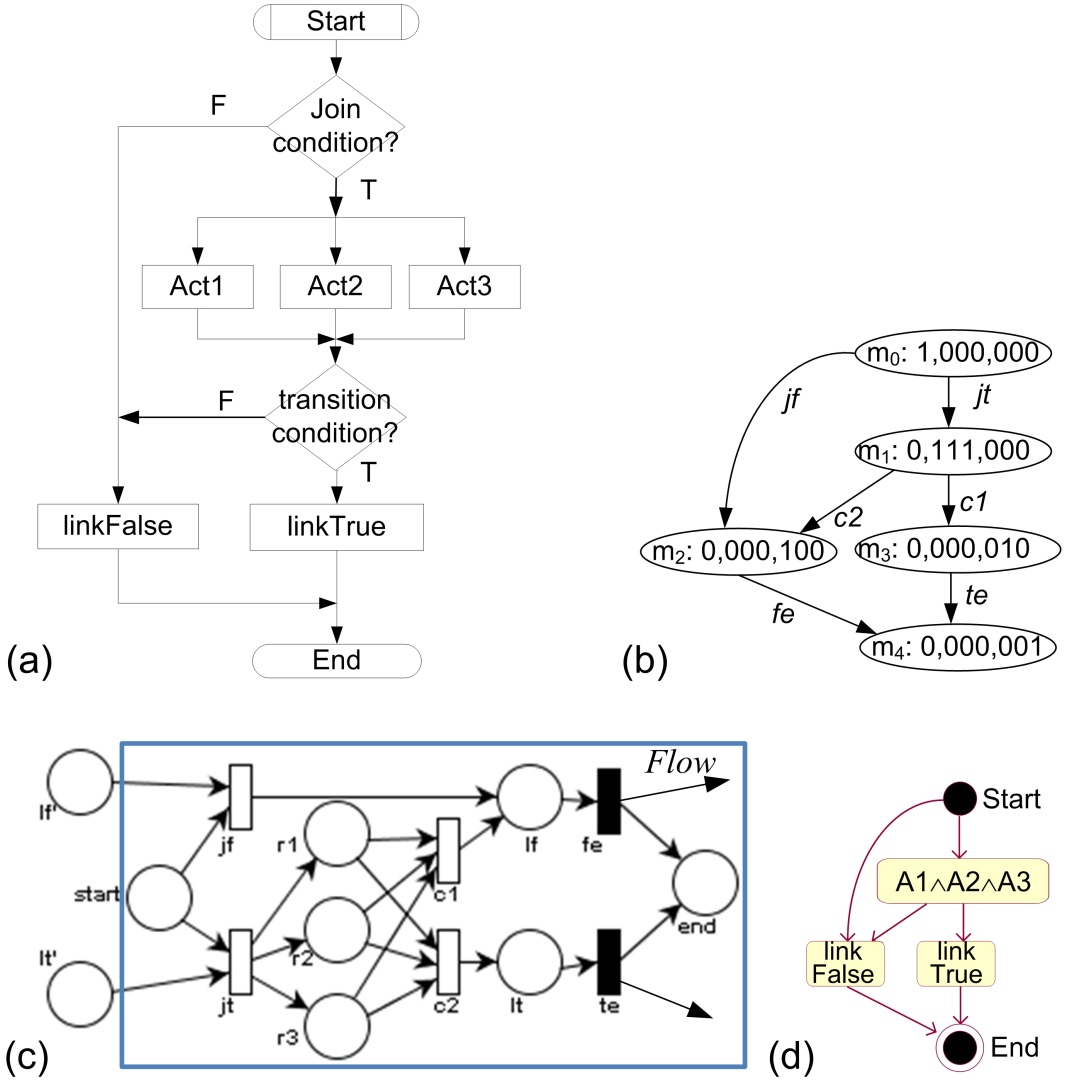


Figure 4.11 *Flow* construct. (a) control flow; (b) reachability graph; (c) Petri net; (d) statechart.

4.4.7 ForEach

The syntax of ForEach is:

forEach ::= "<forEach" foreach-attributes ">"U(standard-elements, startCounterValue, final-CounterValue, O(completionCondition)) Scope "</forEach>"

foreach-attributes ::= U(counterName, parallel, standard-attributes)

The ForEach construct can be sequential or parallel depending on the value of the *parallel* attribute. The operational semantics for sequential and parallel are different, hence we model them separately.

4.4.7.1 Sequential ForEach

The control flow is illustrated in Figure 4.12. If the number of repetitions has not been reached, the scope is re-visited. Otherwise, the loop terminates and the control flow proceeds to the transition condition evaluation. The operational semantics of sequential $ForEach$ is the same with $While$ activity. We design the probabilities of "staying in the loop" and "exiting the loop" so that the mean number of iterations is equal to the predefined number of iterations.

We have $PN_{seq\_foreach} = PN_{while}$.

4.4.7.2 Parallel ForEach

Depending on the values of <startCounterValue> and <finalCounterValue>, there are $(finalCounterValue - startCounterValue + 1)$ instances of $< scope >$. Here we assume there are 3 instances of <scope> to demonstrate our concepts.

CompletionCondition can be used in the parallel *ForEach* activity. When it is used, the *completionCondition* is evaluated upon the completion of each instance. If it is evaluated to true, then all the running instances are terminated. The control flow is demonstrated in Figure 4.13.

As stated in the WS-BPEL 2.0 speicification (81), "In essence an implicit <*flow*> is dynamically created with N+1 copies of the <*ForEach*>>'s enclosed <*scope*> activity as children." Therefore,

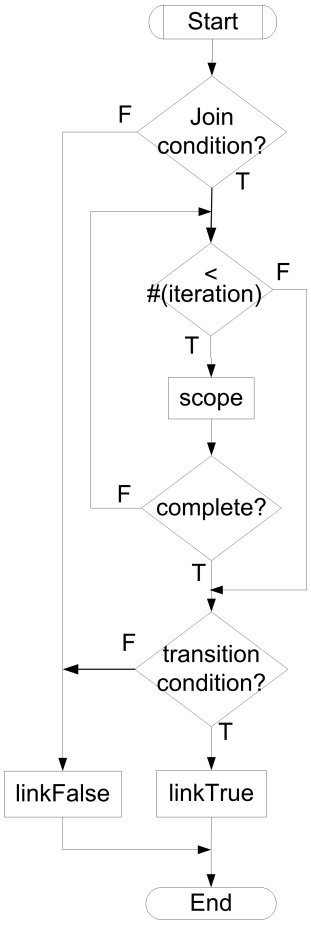


Figure 4.12 Control flow for sequential *ForEach* construct.

1. When the *completionCondition* is not specified, our transformation for parallel *ForEach* follows the rule for *Flow* construct. The hierarchical composition of the $ForEach$ activity also follows the rule for *Flow* construct.

   $PN_{par\_foreach} = PN_{flow}$

   – $1 \leq i \leq n$, *where n is the number of repetitions decided by (finalCounterValue - startCounterValue + 1).*

   – $\{r_i\}$ *is the set of places for "scope" instances.*

   Proof skipped.

2. When the *completionCondition* is specified, any instance completion can terminate all

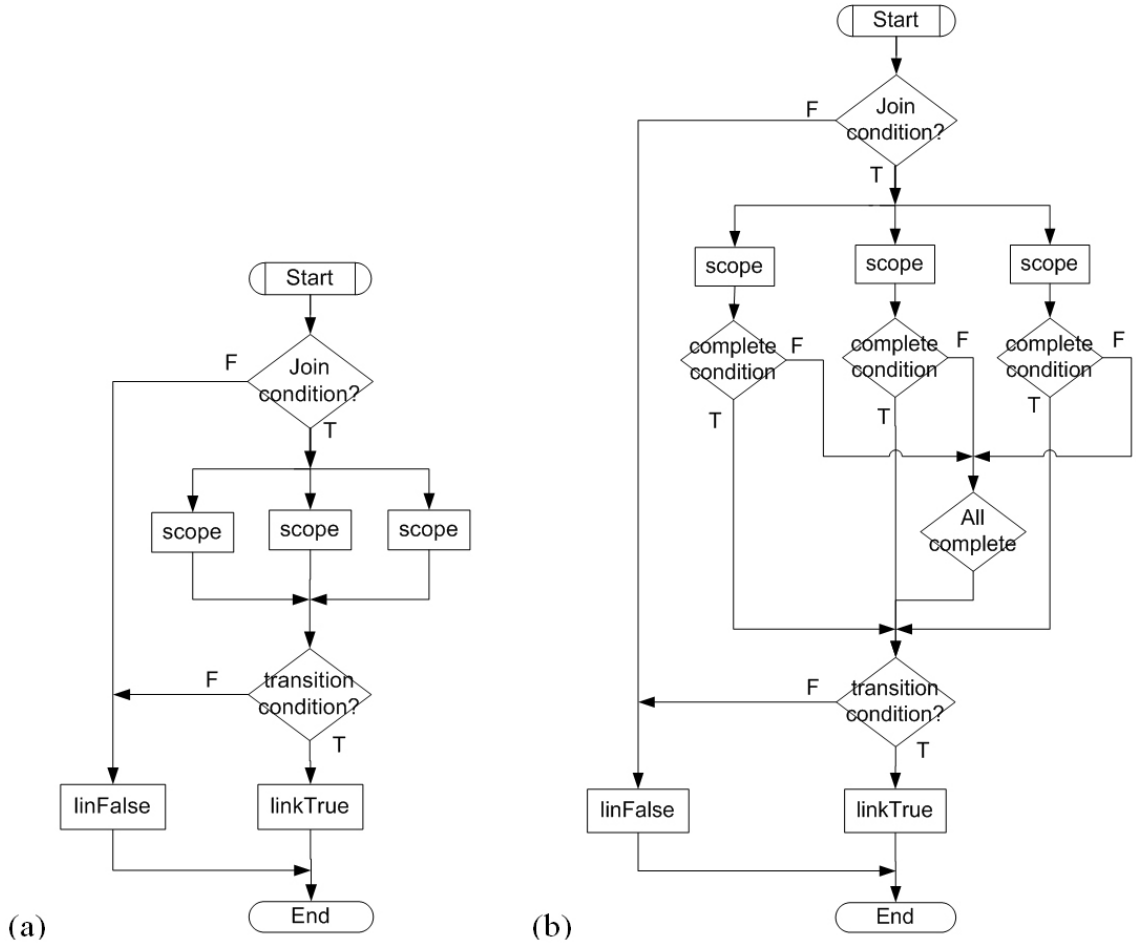


Figure 4.13 Control flow for parallel *ForEach* construct. (a) without *completionCondition*; (b) with *completionCondition*.

other running instances. During atomic transformation, since all the instances are executed concurrently and the execution of each instance is represented by a single place in the Petri net, There is no difference between this case and the previous case when *completionCondition* is absent. Therefore, we generate the same Petri net for this case and "Flow" activity. However, when connecting component activities to the higher level structured activities, rules are different for "Flow" and "parallel ForEach with completionCondition specified", as described in Algorithm 4. Details are proved in section 4.6.1.2.

We have

$PN_{par\_foreach} = PN_{flow}$

– $1 \leq i \leq n$, *where n is the number of repetitions decided by (finalCounterValue - startCounterValue + 1).*

– $\{r_i\}$ *is the set of places for "scope" instances.*

Proof skipped.

4.4.8 Scope

Different from other structured activities, "Scope" is used to enclose an activity (that can be a basic activity or a structured activity) with different handlers, including eventHandlers, faultHandlers, and terminationHandler, etc.

The syntax of Scope is:

scope ::= "<scope" scope-attributes ">" scope-elements "</scope>"

scope-attributes ::= U(standard-attributes, O(isolated), exitOnStandardFault)

scope-elements ::= U(standard-elements, O(partnerLink), O(messageExchanges), O(variables), O(correlationSets), #(handlers), Activities)

We model it as in Figure 4.14.

$PN_{scope} = (P, T, IN, OUT, R, M_0)$, where

$P = \{start, r1, hdlg, lf, lt, end\}$

$T = \{jf, jt, e1, c1, c2, h1, h2, fe, te\}$

$IN = \{(start, jf), (start, jt), (r1, c1), (r1, c2), (r1, e1), (hdlg, h1), (hdlg, h2), (lf, fe), (lt, te)\}$

$OUT = \{(jf, lf), (jt, r1), (c1, lf), (c2, lt), (e1, hdlg), (h1, lf), (h2, lt), (fe, end), (te, sf)\}$

$R = \{r_{jf}, r_{jt}, r_{c1}, r_{c2}, r_{e1}, r_{h1}, r_{h2}\}$ *(firing rates for timed transitions)*

$M_0 = \{[start]\}$.

Proof skipped.

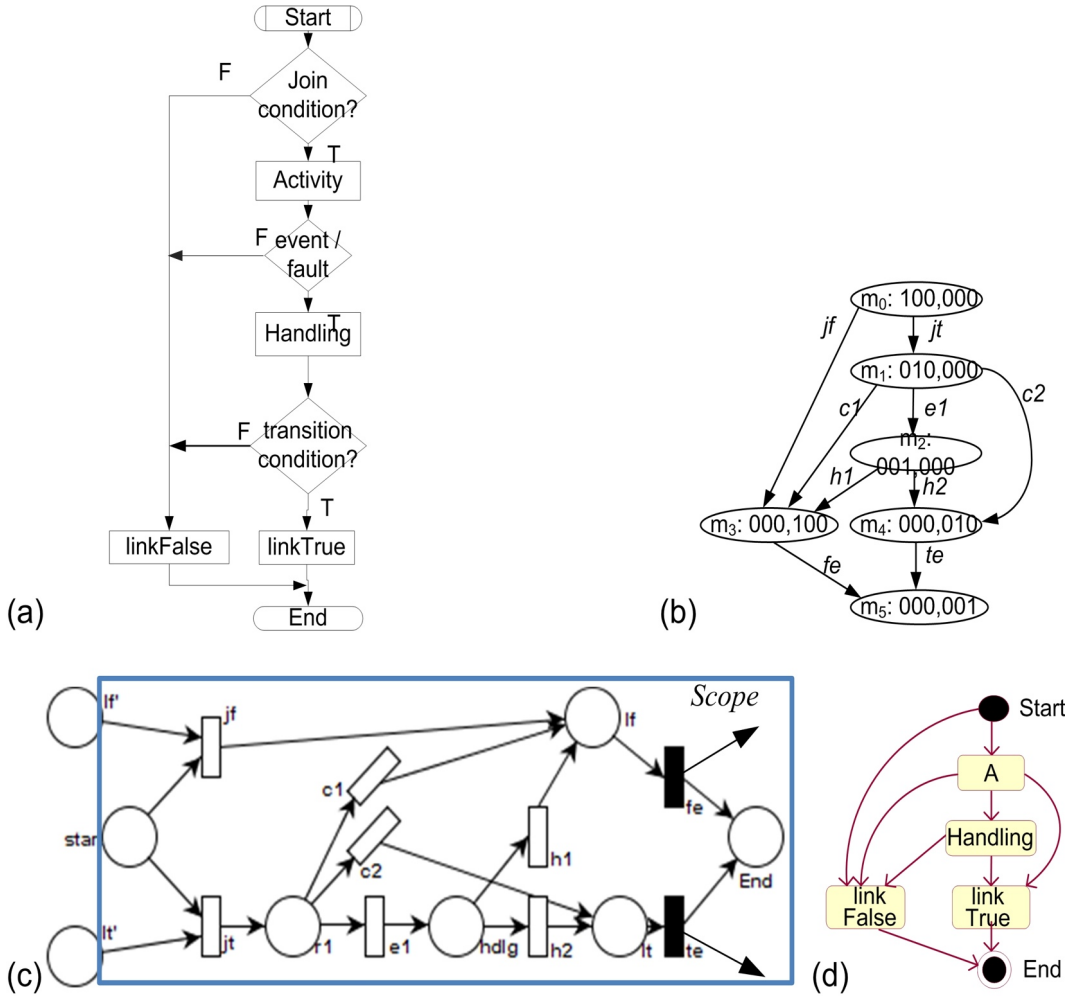


Figure 4.14 *Scope* construct. (a) control flow; (b) statechart; (c) Petri net; (d) reachability graph.

4.5 Transformation of Composition

4.5.1 Algorithms

A business process described in WS-BPEL is an XML document, which has a tree structure. Therefore, we can parse the WS-BPEL description into a tree T of WS-BPEL constructs, abbreviated as *construct tree*. Each node in this tree, abbreviated as *construct node*, is an activity that can be basic (having no child) or structured (with children). Algorithm 1 outlines the overall transformation process.

The construct tree T discussed here carries the same semantics of WS-BPEL. The ordering

of child activities is the same with the one described in the business process. For example, a "Sequence" activity may have 5 or more child activities. The 5 child activities are sibling nodes in the construct tree. They are placed in the same order as in the WS-BPEL process.

In WS-BPEL, the composition is realized through structured activities. In our transformation, this composition is the process to compose child nodes into the parent node in the construct tree. The operation of composition, denoted as $\circ$, is realized in two ways:

- Sequential composition of activities, denoted as $\oplus$. In transformation, the sequential composition is performed as concatenation, i.e., we concatenate the component Petri nets together.

- Hierarchical composition, denoted as $\odot$ (for example, composing a *While* activity using component activities). In transformation, $\odot$ is performed as insertion. More specifically, we generate a Petri net for the parent node first, and then insert the child Petri nets into the parent Petri net. Details are shown in Algorithm 4.

Root Activity.

The root node of a construct tree T is a *Sequence* activity.

Let $\{T_1, T_2, \ldots, T_n\}$ be the set of T's children, and $\{PN_1, PN_2, \ldots, PN_n\}$ be the set of Petri nets corresponding to the child nodes.

Then $PN_T = PN_1 \oplus PN_2 \oplus \ldots \oplus PN_n$. Note that each T_i may be hierarchical, so does PN_i.

Algorithm 1 depicts the overall process of transformation. This algorithm takes a WS-BPEL document as input and outputs a Petri net.

Algorithm 1. *PN BPEL2PN(BPEL Bprocess)*

Begin

Parse *Bprocess* into construct tree T

$PN_{Bprocess}$ = ProcessNode(T)

return $PN_{Bprocess}$

End

Algorithm 2 describes how we process the construct tree. This recursive algorithm traverses the construct tree in depth-first order. Consequently, the Petri net is generated hierarchically.

When T is a basic activity, it is represented as a leaf node in the construct tree. In this case, Lp is an empty set.

Algorithm 2. *PN ProcessNode(node T)*

Begin

Let Lp be empty list

For each T_i child of T **Do**

PN_i = ProcessNode(T_i)

Add PN_i to Lp

Generate PN_T for T with Lp using StructuredAct(T, Lp)

/*Lp is $\emptyset$ when T has no child*/

Return PN_T

End

The composition process, $\circ$, is depicted in function *StructuredAct*(T, Lp) (Algorithm 3). As described at the beginning of this section, we process composition in two ways: concatenation and insertion. Concatenation is described in Algorithm 3. The insertion operation, $\odot$, is performed using function *InsertPN* that is displayed in Algorithm 4.

Note that when we process "Flow" activity (in Algorithm 3) we transform the "Flow" activity regardless its enclosed control links first, and then add the control links to the generated Petri net by invoking the *ProcessLink* function (shown in Algorithm 5). As stated in section 4.3, this is because that control links may cross sequential activity boundaries.

Algorithm 3. *PN StructuredAct(node T, List Lp)*

Begin

Switch T

Case "Basic":

Generate a Petri net PN_T for T according to PN_{BAWL}, PN_{BANL}, or PN_{exit}

Case "Sequence": /*operation $\oplus$*/

Concatenate Lp sequentially to PN_T

/*merge the *end* place of PN_{i-1} with the *start* place of PN_i*/

Case "If", "Pick":

Generate a Petri net PN_T according to PN_{if}

$/ * |r_i| = |Lp| * /$

Replace places $\{r_i\}$ in PN_T with Lp using $InsertPN(PN_T, Lp)$

Case "While", "Sequential ForEach":

Generate a Petri net PN_T according to PN_{while}

Replace place $r1$ in PN_T with Lp using $InsertPN(PN_T, Lp)$

Case "RepeatUntil":

Generate a Petri net PN_T according to $PN_{RepeatUntil}$

Replace place $r1$ in PN_T with Lp using $InsertPN(PN_T, Lp)$

Replace places r_i in PN_T with Lp using $InsertPN(PN_T, Lp)$

Case "Flow":

Generate a Petri net PN_T according to PN_{flow}

Replace places $\{r_i\}$ in PN_T with Lp using $InsertPN(PN_T, Lp)$

ProcessLink($Bprocess$, PN_T)

Case "Parallel ForEach":

Generate a Petri net PN_T according to PN_{flow}

Replace places $\{r_i\}$ in PN_T with Lp using $InsertPN(PN_T, Lp)$

Case "Scope":

Generate a Petri net PN_T according to PN_{scope}

Replace place $r1$ in PN_T with Lp using $InsertPN(PN_T, Lp)$

End Switch

Return PN_T

End

Algorithm 4. *PN InsertPN(PN PN_T, List Lp)*

Begin

Switch PN_T

Case "If":

For each PN_i in Lp **Do**

Remove place r_i from PN_T

Remove links $(cond_i, r_i)$, (r_i, c_{i1}), and (r_i, c_{i2}) from PN_T

Add links $(cond_i, start_i)$, (end_i, c_{i1}) and (end_i, c_{i2}) to PN_T

$/*start_i$ and end_i are the "*start*" and "*end*" places of PN_i, respectively$*/$

Case "While", "Sequential ForEach":

Pop PN_1 from Lp

$/*$ *While* activity has one child node: basic activity or *Sequence* activity $*/$

Remove links $(jt, r1)$, $(r1, loop)$, $(loop, r1)$, $(r1, c1)$, and $(r1, c2)$ from PN_T

Remove place $r1$ from PN_T

Add links $(jt, start_1)$, $(end_1, c1)$ and $(end_1, c2)$ to PN_T

$/*start_1$ and end_1 are the "*start*" place and "*end*" place of PN_1, respectively$*/$

Case "RepeatUntil":

Pop PN_1 from Lp

$/*$ *RepeatUntil* activity has only one child node

Remove links $(jt, r1)$, $(r1, loop)$, $(loop, r1)$, $(r1, c1)$, and $(r1, c2)$ from PN_T

Remove place $r1$ from PN_T

Add links $(jt, start_1)$, $(end_1, c1)$ and $(end_1, c2)$ to PN_T

$/*start_1$ and end_1 are the "*start*" place and "*end*" place of PN_1, respectively$*/$

Case "Pick":

For each PN_i in Lp **Do**

Remove place r_i from PN_T

Remove links $(cond_i, r_i)$, (r_i, c_{i1}), and (r_i, c_{i2}) from PN_T

Add links $(cond_i, start_i)$, (end_i, c_{i1}) and (end_i, c_{i2}) to PN_T

/*$start_i$ and end_i are the "*start*" and "*end*" places of PN_i, respectively*/

Case "Flow":

For each PN_i in Lp **Do**

Remove place r_i from PN_T

Remove links (jt, r_i), (r_i, c_{i1}), and (r_i, c_{i2}) from PN_T

Add links $(jt, start_i)$, (end_i, c_{i1}) and (end_i, c_{i2}) to PN_T

/*$start_i$ and end_i are the "*start*" and "*end*" places of PN_i, respectively*/

Case Parallel "ForEach" without *completionCondition* specified:

For each PN_i in Lp **Do**

Remove place r_i from PN_T

Remove links (jt, r_i), (r_i, c_{i1}), and (r_i, c_{i2}) from PN_T

Add links $(jt, start_i)$, (end_i, c_{i1}) and (end_i, c_{i2}) to PN_T

/*$start_i$ and end_i are the "*start*" and "*end*" places of PN_i, respectively*/

Case Parallel "ForEach" with *completionCondition* specified:

For each PN_i in Lp **Do**

Remove place r_i from PN_T

Remove links (jt, r_i), (r_i, c_{i1}), and (r_i, c_{i2}) from PN_T

Add links $(jt, start_i)$, (end_i, c_{i1}) and (end_i, c_{i2}) to PN_T

/*$start_i$ and end_i are the "*start*" and "*end*" places of PN_i, respectively*/

For each transition in $\{c_{ik} | c_{ik} \in T_i$ and c_{ik} points to $e_i\}$ **Do**

Update the name of transition c_{ik} to ccf_{ik}

/*Now add extra transition cct_{ik} and marking-dependent arcs*/

Add transition cct_{ik}

Add arcs $(P_j - \{e_j\}, cct_{ik})$, where $j \neq i$ and P_j is the set of places in PN_j

Set the arc cardinality of (p_j, cct_{ik}) to be μ_{p_j}

Let p_i = the input place of transition ccf_{ik}, add arc (p_i, cct_{ik})

Add arcs $(cct_{ik}, \{e_j\}_{1 \leq j \leq n})$

Case "Scope":

Pop PN_1 from Lp

$/*$ *Scope* activity has only one child node

Remove links $(jt, r1)$, $(r1, e1)$, $(r1, c1)$, and $(r1, c2)$ from PN_T

Remove place $r1$ from PN_T

Add links (jt, $start_1$), (end_1, $e1$), (end_1, $c1$) and (end_1, $c2$) to PN_T

$/*start_1$ and end_1 are the "*start*" and "*end*" places of PN_1, respectively$*/$

End Switch

Return PN_T

End

In WS-BPEL, each control link has a unique ID, called "name", and specified source activity and target activity. Therefore, it is possible to find and connect the corresponding source Petri net and target Petri net for a specific link.

For each join condition, let l be the number of incoming links. There are totally $2^l - 1$ "join false" transitions and 1 "join true" transition jt, as discussed in section 4.3. We number all the join transitions according to their incoming link status. These numbers can be converted into bit-strings. Each of such a string has l bits and each bit represents a link status. For example, for 6'th join transition of join condition d, the bit-string is [0, 0, ..., 1, 0, 0]. That means only $link_3 = true$. All other links are $false$. In this case, we should connect lt_3 to jf_{d6}, and connect $\{lf_k | 1 \leq k \leq l \text{ and } k \neq 3\}$ to jf_{d6}. Similarly, the bit-string for jt_d is [1, 1, ..., 1], which is converted from integer $2^l - 1$. That means all the incoming links are $true$. We should connect $\{lt_k | 1 \leq k \leq l\}$ to jt_d.

Following the above rules, we use Algorithm 5 to process control links.

Algorithm 5. *PN ProcessLink(BPEL T, PN pnt)*

Begin

For each control link (s, t) in PN_T **Do**

/* Firstly, connect from source activities to link status places. */

/ * fe_s and te_s are the transitions pertaining to the source Petri net. */

Create places lf_{st} and lt_{st}

Add arc (te_s, lt_{st})

If the source activity specifies *transitionCondition*, **Then**

Add arc (fe_s, lf_{st})

For each (join condition) **Do**

/* Secondly, connect link status places to the target activities.*/

Let l be the number of incoming links for this join condition

For i from 0 to $2^l - 1$ **Do**

/* Connect each join transition JT_i. $JT_i = jt$ when $i = 2^l - 1$; Otherwise $JT_i = jf_i$ */

Convert i to bit-string B, B contains l bits

For j from 1 to l **Do**

/*Connect each incoming link to JT_i. */

/* lf_j and lt_j are the places representing link status for the $j'th$ incoming link. */

/* In the Petri net, lf_j and lt_j are represented as lf_{st} and lt_{st}, where t is the Petri net holding the current join condition */

If $B_j = 0$, **Then**

Add arc (lf_j, JT_i)

Else

Add arc (lt_j, JT_i)

If the target activity is a *While* construct, **Then**

/* $fc1_t$, and $fc2_t$ are the transitions pertaining to the target Petri net.*/

For j from 1 to l **Do**

Add arcs $(lt_j, fc1_t)$ and $(lt_j, fc2_t)$

End

4.5.2 Examples

There are plenty of tools available for analyzing and simulating Petri nets, through which different properties can be verified. Our work of formalizing the transformation from WS-BPEL to Petri net bridges the business process design language and the analytic models. This makes it easier for service designers to verify their design, conduct performance analysis, and carry out impact analysis, etc. In the following, we use two examples to illustrate how service composition can be handled using Petri net. These examples are extracted from the SmartHome project at Iowa State University (47).

4.5.2.1 The SeeMovie example

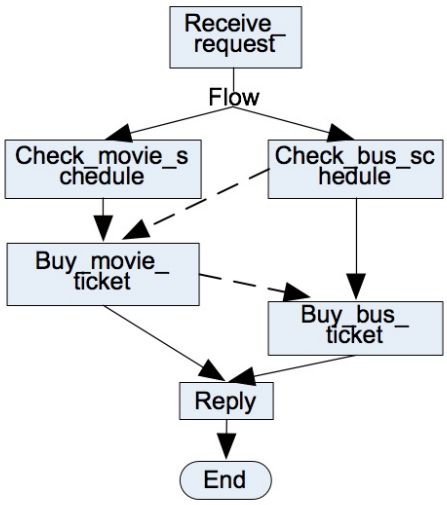


Figure 4.15 *SeeMovie* WS-BPEL process.

The first example describes the scenario of going out to see a movie. The WS-BPEL process describing the behavioral model is shown in Appendix B. When the elder resident invokes the SeeMovie service, a flow construct is executed to check movie/bus schedule and purchase tickets. As shown in the Figure 4.15, there are two synchronizing links between the two parallel sequences. One is from *Invoke_BusSchedule* to *Invoke_Ticket* since one can only decide which bus to take when the movie schedule is set. The other synchronizing link goes

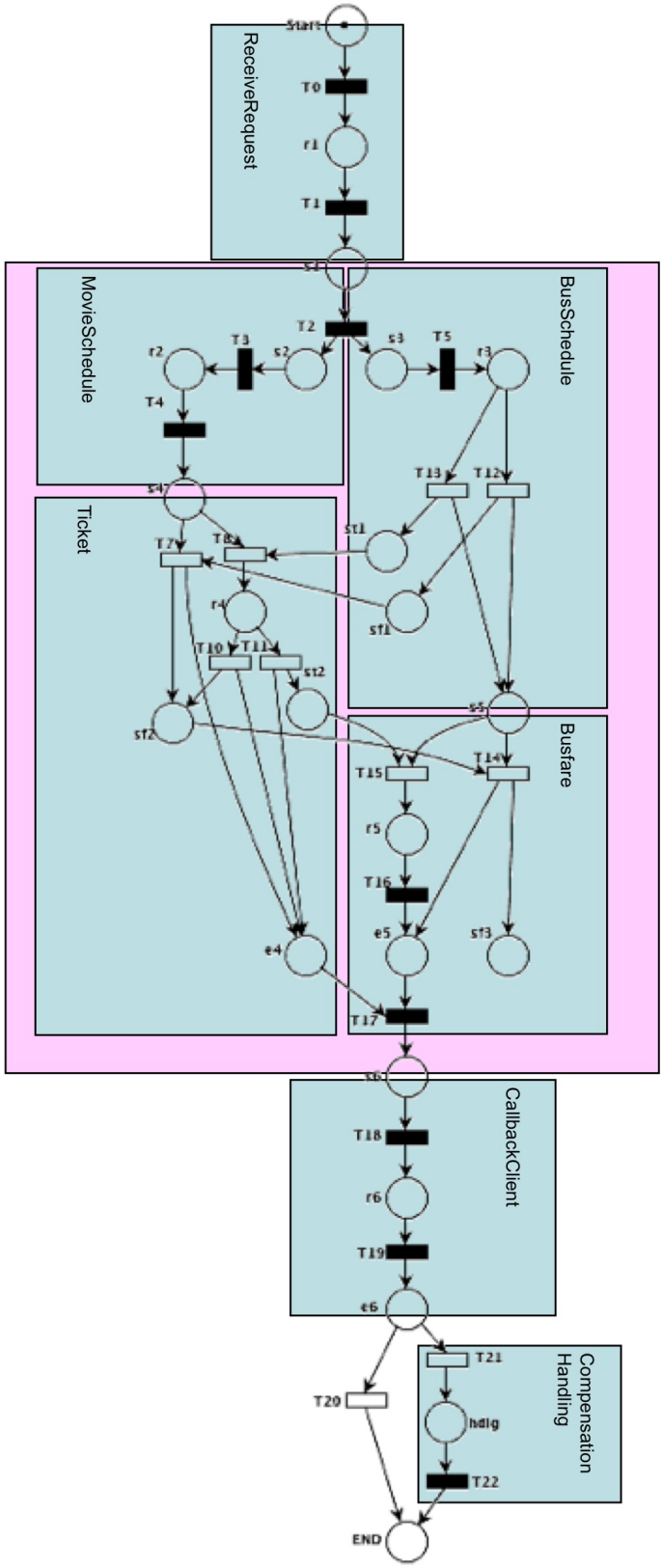


Figure 4.16 $PN_{seemovie}$

from the *Invoke_MovieTicket* service to the *Invoke_BusFare*, which decides that only after the movie ticket is purchased do we purchase bus tickets.

The whole procedure is enclosed in a "Scope" activity. This "Scope" activity has a event handler. The event handler catches *cancel* messages from the client. When a *cancel* message arrives, a compensate activity reverses the transactions (returning the tickets, etc.) and then terminates the whole process. This scenario can be expanded to include other transportation methods such as a taxi. Here we simplify our example to demonstrate the merits of composition.

Figure 4.15 displays this WS-BPEL process. Figure 4.16 depicts its Petri net presentation. The "Compensation Handling" illustrates the event handler associated with the "Scope" activity. The analysis of Figure 4.16 shows this Petri net is safe and livelock free.

4.5.2.2 The FireHazard example

The second example is used to illustrate flow and IF constructs. The WS-BPEL process describing the behavioral model is shown in Appendix C. It describes the composite service of fire hazard processor. In an efficiency apartment within a smart apartment complex, the digital smoke detector senses an unusual amount of smoke in the air, which immediately triggers the service of *FireHazardProcessor*. This service initiates the fire alarm and the sprinkler system as part of fire emergency service and makes 911 emergency calls. The next step is to check fire status through smoke detector, chemical sensors or temperature sensors. If fire is assured to be eliminated, *FireHazardProcessor* service ends itself; otherwise this loop continues till the fire is out. The WS-BPEL process of this service is depicted in Figure 4.17 and its generated Petri net is shown as in Figure 4.18.

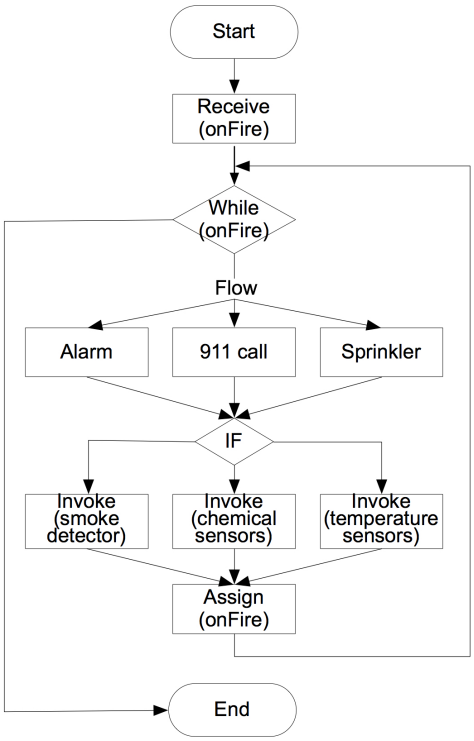


Figure 4.17 *FireHazard* WS-BPEL process.

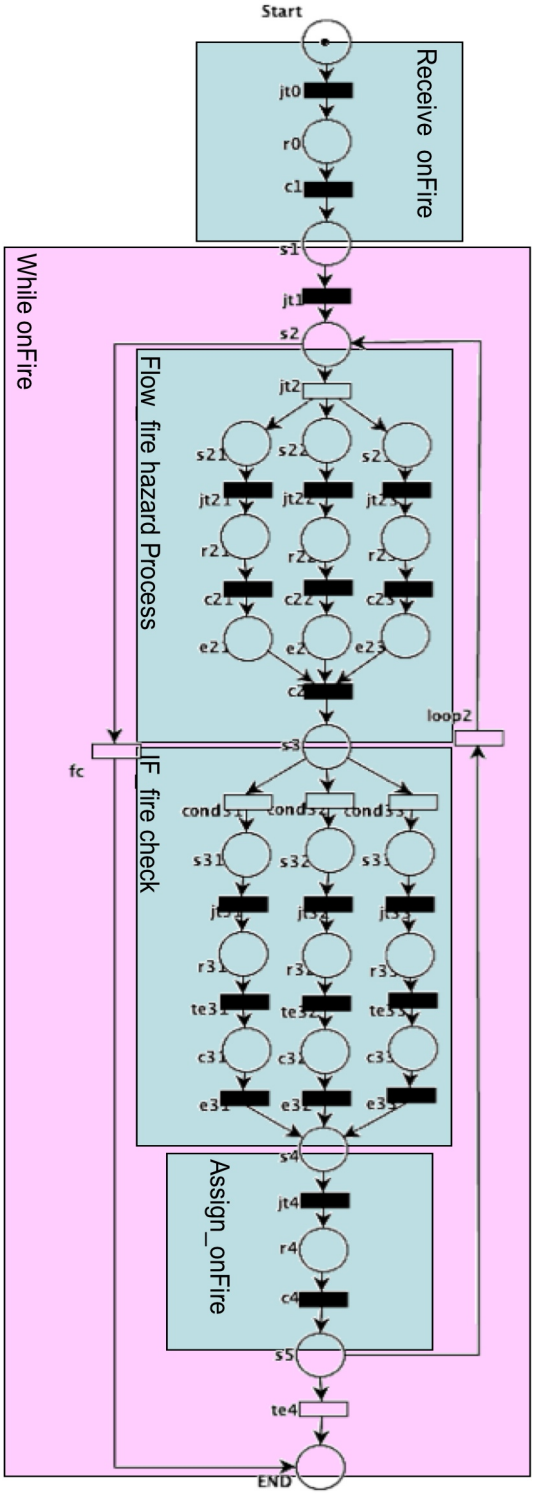


Figure 4.18 $PN_{firehazard}$.

4.6 Correctness of Transformation

In section 4.4 we have proved that the atomic transformation preserves the original system behavior (the soundness). Now we demonstrate that the soundness is preserved during the composition process.

4.6.1 Correctness of atomic composition

"Atomic composition" means that only one step composition is performed, such as composing two sequential activities and composing one structured activity using its component activities. Section 4.5 lists two ways of composition: sequential composition and hierarchical composition. In the transformation presented in this thesis, the sequential composition is processed as concatenation, and the hierarchical composition is processed as insertion.

Note that the sequential composition may still involve control links if both activities of the sequential composition belong to a "Flow" activity. However, we do not process control links unless we have connected all the component Petri nets of a "Flow" activity, as stated above in Algorithm 4.

Below we show that atomic sequential composition and atomic hierarchical composition are correct.

4.6.1.1 Correctness of atomic sequential composition

Let $PN_1 = (P_1, T_1, IN_1, OUT_1, R_1, M_{0_1})$, $PN_2 = (P_2, T_2, IN_2, OUT_2, R_2, M_{0_2})$.

Then $PN_T = PN_1 \bigoplus PN_2 = (P, T, IN, OUT, R, M_0)$, where

$P = P_1 \bigcup P_2 - \{end_1\}$

$T = T_1 \bigcup T_2$

$IN = IN_1 \bigcup IN_2$

$OUT = OUT_1 \bigcup OUT_2 \bigcup \{(t, start_2) | \exists t \in T_1 \text{ and } (t, end_1) \in OUT_1\}$

$\quad -\{(t, end_1) | \exists t \in T_1 \text{ and } (t, end_1) \in OUT_1\}$

$R = R_1 \bigcup R_2$

$M_0 = \{[start_1]\}$

Let G_1 and G_2 be the reachability graphs of PN_1 and PN_2.

Let S_1 and S_2 be the statecharts of T_1 and T_2. We have:

$G_1 = (V_1, E_1)$,

$S_1 = (A_1, B_1)$,

Bijection function $f : V_1 \rightarrow A_1$

G_1 and S_1 are isomorphic, i.e., $(c, d) \in E_1 \Leftrightarrow (f(c), f(d)) \in B_1$

$G_2 = (V_2, E_2)$,

$S_2 = (A_2, B_2)$,

Bijection function $t : V_2 \rightarrow A_2$

G_2 and S_2 are isomorphic, i.e., $(x, y) \in E_2 \Leftrightarrow (t(x), t(y)) \in B_2$

Obviously, the reachability graph of T, G, is the concatenation of G_1 and G_2. $G = (V, E)$, where

$V = V_1 \bigcup V_2$

$[end_1] = [start_2]$, where $[end_1]$ is the end marking of G_1 and $[start_2]$ is the start marking of G_2

$E = E_1 \bigcup E_2$

$V_1 \bigcap V_2 = [start_2]$

$E_1 \bigcap E_2 = \emptyset$

Similarly, define the statechart of T as $S = (A, B)$, where

$A = A_1 \bigcup A_2$

$end_1 = start_2$, end_1 is the end state of S_1 and $start_2$ is the start state of S_2

$B = B_1 \bigcup B_2$

$A_1 \bigcap A_2 = start_2$

$B_1 \bigcap B_2 = \emptyset$

Theorem 2. *Concatenating sequential Petri nets preserves the soundness of atomic transformation, i.e., $G \cong S$(G and S are defined as above).*

Proof. Define a bijection function $h : V \rightarrow A$ as

$$h(x) = \begin{cases} f(x) \text{ if } x \in V_1 \\ t(x) \text{ if } x \in V_2 \end{cases}$$

Because B_1 and B_2 are disjoint,

$(h(x), h(y)) \in B_1$

$\Leftrightarrow x, y \in V_1$

$\Leftrightarrow (h(x) = f(x)) \wedge (h(y) = f(y))$

(4.3)

Similarly,

$$(h(x), h(y)) \in B_2 \Leftrightarrow (h(x) = t(x)) \wedge (h(y) = t(y)) \tag{4.4}$$

We need to prove that

$(x, y) \in E \Leftrightarrow (h(x), h(y)) \in B$.

$(h(x), h(y)) \in B$

$\Leftrightarrow (h(x), h(y)) \in B_1$ or $(h(x), h(y)) \in B_2$

$\Leftrightarrow (f(x), f(y)) \in B_1$ or $(t(x), t(y)) \in B_2$ by (4.3) and (4.4)

$\Leftrightarrow (x, y) \in E_1$ or $(x, y) \in E_2$

$\Leftrightarrow (x, y) \in E$ □

From Theorem 2, we can see that after sequential composition, the transformed Petri net and the original WS-BPEL description still have isomorphic state transition models.

Example.

Below we use an example to illustrate this process. Figure 4.19 demonstrate the sequential composition of two basic activities, T_1 and T_2.

The isomorphism between the startchart and the reachability graph shown in Figure 4.19 indicates that our transformation of composing two sequential basic activities preserves the original system behavior. In fact, the transformation of composing any two activities, either basic or structured, will not change the original system state transition. This has been proved in theorem 2.

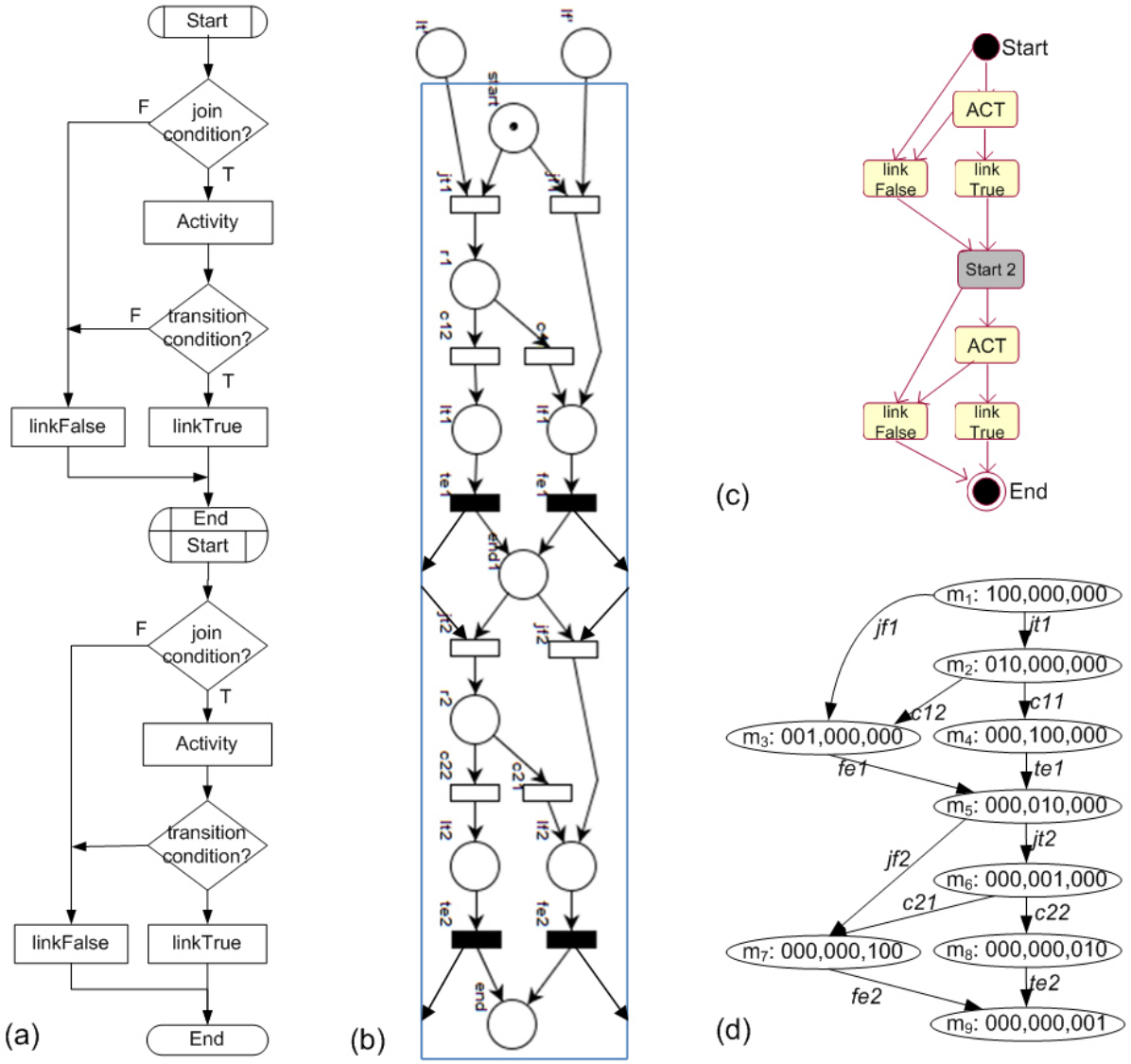


Figure 4.19 Composing two basic activities (a) Control flow; (b) Petri net; (c) Statechart; (d) Reachability graph.

4.6.1.2 Correctness of atomic hierarchical composition

In a hierarchical state model, a certain state can be expanded to a state model of a lower-level system. In other words, a component state transition system is represented as a single state in the higher-level system. Our hierarchical composition follows the same rule.

However, the situation is more complex when the higher-level system has concurrency (parallel execution). In this case, the state model of the higher-level system is not a simple composition of its component state models as described above. One state in the higher-level state model represents not only the state of one component system, but the state of all the concurrent component systems. In fact, the state transition of the concurrent execution is a product of all the concurrent state models. If the concurrent execution has synchronization dependency, the situation is even more complicated.

Below we study the four cases separately:

(1) hierarchical composition without concurrency;

(2) hierarchical composition with concurrency but no synchronization;

(3) Parallel $ForEach$ when $completionCondition$ is specified, which is a special case of (2); and

(4) hierarchical composition with concurrency and synchronization.

In our transformation from WS-BPEL to Petri net, "$Flow$" construct belongs to case (2) and case (4). Parallel "$ForEach$" when $completionCondition$ is not specified belongs to case (2). Parallel "$ForEach$" when $completionCondition$ is specified is a special case of (2). The concurrent execution of (3) is the same with the one of (2). However, because we add extra transitions and marking-dependent arcs (18), the ending of the concurrent execution in (2) is different from the one in (3). Therefore, we take (3) out as a special case. All other hierarchical composition, such as $While$, $Sequential\ ForEach$, If, and $Scope$, belong to case (1).

4.6.1.3 Correctness of atomic hierarchical composition without concurrency.

T is the structured activity that is composed using T_a. PN_P is the Petri nets generated for T according to its activity type during atomic transformation. PN_a is the Petri net modelling

T_a. Following algorithm shown in Algorithm 4, PN_a is inserted into PN_P to replace the $r1$ place. The Petri net after the insertion is denoted as PN_T. We have

$PN_P = (P_P, T_P, IN_P, OUT_P, R_P, M_{0_P})$,

$PN_a = (P_a, T_a, IN_a, OUT_a, R_a, M_{0_a})$.

Then $PN_T = PN_P \bigodot PN_A = (P, T, IN, OUT, R, M_0)$, where

$P = P_P \bigcup P_a - \{r1_P\}$

$T = T_P \bigcup T_a$

$IN = IN_P \bigcup IN_a \bigcup \{end_a, \{tran_P\}\} - \{r1_P, \{tran_P\}\}$, where $\{tran_P\}$ is the set of $r1_P$'s output transitions.

$OUT = OUT_P \bigcup OUT_a \bigcup \{\{tran_P\}, start_a\} - \{\{tran_P\}, r1_P\}$, where $\{tran_P\}$ is the set of $r1_P$'s input transitions.

$R = R_P \bigcup R_a$

$M_0 = \{[start_P]\}$.

Let G_P and G_a be the reachability graphs of PN_P and PN_a.

Let S_P and S_a be the statecharts of T_P and T_a. We have:

$G_P = (V_P, E_P)$,

$S_P = (A_P, B_P)$,

$G_a = (V_a, E_a)$,

$S_a = (A_a, B_a)$,

From the atomic transformation, we have $G_P \cong S_P$ and $G_a \cong S_a$.

Define a bijection function $t : V_P \rightarrow A_P$,

$$(c, d) \in E_P \Leftrightarrow (t(c), t(d)) \in B_P \tag{4.5}$$

Bijection function $f : V_a \rightarrow A_a$,

$$(x, y) \in E_a \Leftrightarrow (f(x), f(y)) \in B_a \tag{4.6}$$

More specifically, we have

$f([start_a]) = start_a$,

$f([end_a]) = end_a$,

$t([r1_P]) = r_P$, where $[r1_P] \in V_P$ and $[r1_P]$ is the state to be replaced hierarchically by G_a; $r_P \in A_P$ and r_P is the state to be replaced hierarchically by S_a.

Following the properties of hierarchical state model, the reachability graph of PN_T is:

$G = (V, E)$, where

$$V = V_P \bigcup V_a - \{[r1_P]\}$$

$$E = E_P \bigcup E_a$$
$$\bigcup\{([end_a], [start_a]) \mid ([r1_P], [r1_P]) \in E_P,$$
$$(s_P, [start_a]) \mid \exists s_P \in V_P \text{ and } s_P \neq [r1_P], (s_P, [r1_P]) \in E_P,$$
$$([end_a], t_P) \mid \exists t_P \in V_P \text{ and } t_P \neq [r1_P], ([r1_P], t_P) \in E_P\}$$
$$-\{(s_P, [r1_P]) \mid \exists s_P \in V_P \text{ and } s_P \neq [r1_P], (s_P, [r1_P]) \in E_P,$$
$$([r1_P], t_P) \mid \exists t_P \in V_P \text{ and } t_P \neq [r1_P], ([r1_P], t_P) \in E_P\}$$

Let $E_b = \{([end_a], [start_a]) \mid ([r1_P], [r1_P]) \in E_P,$
$$(s_P, [start_a]) \mid \exists s_P \in V_P \text{ and } s_P \neq [r1_P], (s_P, [r1_P]) \in E_P,$$
$$([end_a], t_P) \mid \exists t_P \in V_P \text{ and } t_P \neq [r1_P], ([r1_P], t_P) \in E_P\}$$

Let $E_{ps} = \{(s_P, [r1_P]) \mid \exists s_P \in V_P \text{ and } s_P \neq [r1_P], (s_P, [r1_P]) \in E_P,$
$$([r1_P], t_P) \mid \exists t_P \in V_P \text{ and } t_P \neq [r1_P], ([r1_P], t_P) \in E_P\}$$

Obviously $E_{ps} \subset E_P$

Then $E = (E_P - E_{ps}) \bigcup E_a \bigcup E_b$

$(E_P - E_{ps}), E_a, E_b$ **are pairwise disjoint.**

Similarly, the statechart of T is

$S = (A, B)$, where

$$A = A_P \bigcup A_a - \{r_P\}$$

$$B = B_P \bigcup B_a$$
$$\bigcup\{(end_a, start_a) \mid (r_P, r_P) \in B_P,$$

$$
\begin{aligned}
&(ss_P, start_a) \mid \exists ss_P \in A_P \text{ and } s_P \neq r_P, (s_P, r_P) \in B_P, \\
&(end_a, tt_P) \mid \exists tt_P \in A_P \text{ and } tt_P \neq r_P, (r_P, tt_P) \in B_P\} \\
-\{&(ss_P, r_P) \mid \exists ss_P \in A_P \text{ and } ss_P \neq r_P, (ss_P, r_P) \in B_P, \\
&(r_P, tt_P) \mid \exists tt_P \in A_P \text{ and } tt_P \neq r_P, (r_P, tt_P) \in B_P\} \\
= &(B_P - B_{ps}) \bigcup B_a \bigcup B_b
\end{aligned}
$$

$(B_P - B_{ps}), B_a, B_b$ are pairwise disjoint.

Theorem 3. *Based on the definitions of G, G_P, G_a, S, S_P, S_a, if $G_P \cong S_P$ and $G_a \cong S_a$, then $G \cong S$.*

Proof. Define a bijection function $h : V \to A$ as

$$h(x) = \begin{cases} t(x) \text{ if } x \in V_P \\ f(x) \text{ if } x \in V_a \end{cases}$$

Because $(B_P - B_{ps}), B_a$ and B_b are pairwise disjoint,

$(h(x), h(y)) \in B$

$\Leftrightarrow$ (1)$(h(x), h(y)) \in (B_P - B_{ps})$

or (2) $(h(x), h(y)) \in B_a$

or (3) $(h(x), h(y)) \in B_b$

$\Leftrightarrow$ (1)$(t(x), t(y)) \in B_P$

and $t(x) \neq r_P$ and $t(y) \neq r_P$

or (2) $(f(x), f(y)) \in B_a$

or (3) $(h(x), h(y)) = (end_a, start_a)$

or $(h(y) = start_a \wedge h(x) \in A_P \wedge h(x) \neq r_P \wedge (h(x), r_P) \in B_P)$

or $(h(x) = end_a \wedge h(y) \in A_P \wedge h(y) \neq r_P \wedge (r_P, h(y)) \in B_P)$

$\Leftrightarrow$ (1)$(x, y) \in E_P$

and $x \neq [r1_P]$ and $y \neq [r1_P]$

or (2) $(x, y) \in E_a$

$$
\begin{aligned}
&\text{or } (3)\ ((x,y) = ([end_a],[start_a]) \\
&\quad \text{or } (y = [start_a] \wedge h(x) \in V_P \wedge h(x) \neq [r1_P] \wedge (x,[r1_P]) \in E_P) \\
&\quad \text{or } (x = [end_a], \wedge h(y) \in V_P \wedge h(y) \neq [r1_P] \wedge ([r1_P],y) \in E_P)) \\
&\Leftrightarrow (1)(x,y) \in (E_P - E_{ps}) \\
&\quad \text{or } (2)\ (x,y) \in E_a \\
&\quad \text{or } (3)\ (x,y) \in E_b \\
&\Leftrightarrow (x,y) \in E
\end{aligned}
$$

□

Theorem 3 ensures that inserting one component state transition models into the parent state transition model preserves the soundness of atomic transformation.

Theorem 4. *Hierarchical composition without concurrency preserves the original system behavior.*

Proof. For "While" and "Sequential ForEach" composition, we have only one state r_P to be replaced. This soundness is proved by Theorem 3.

For "If" and "Pick" compositions, we have multiple child state transition models to insert into the parent model. We can conduct the insertion step by step. Each step of insertion preserves the soundness according to Theorem 3. Therefore, after the insertion process is done, the composed Petri net still preserves the original behavior of the composed activity. □

Example.

In this example, we compose the "While" activity using its child activity, T_a. The control flow and statechart are shown in Figure 4.20.

The transformation follows Algorithm 3. We first generate a Petri net PN_a for its child activity, then create Petri net PN_{while} according to the transformation defined in section 4.4.3, and finally insert PN_a into PN_{while} and update arcs. This process is shown in Figure 4.21. The corresponding reachability graphs are shown in Figure 4.22.

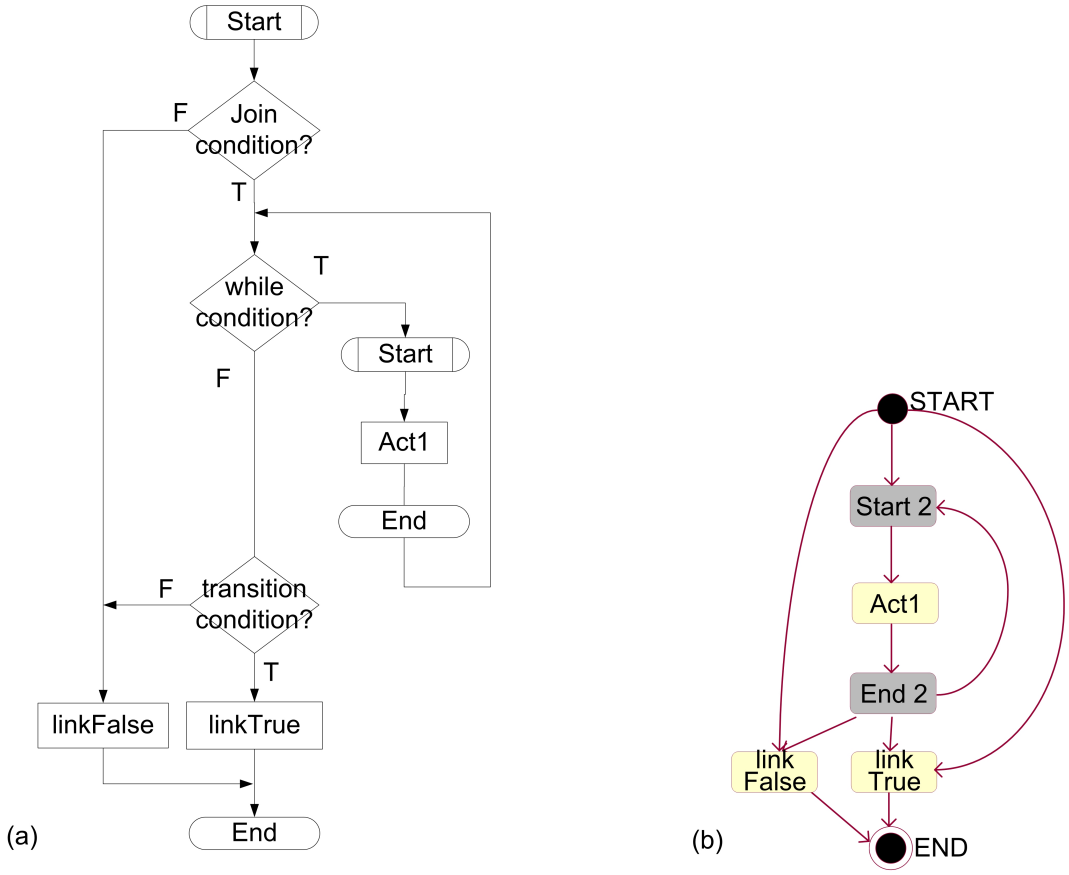
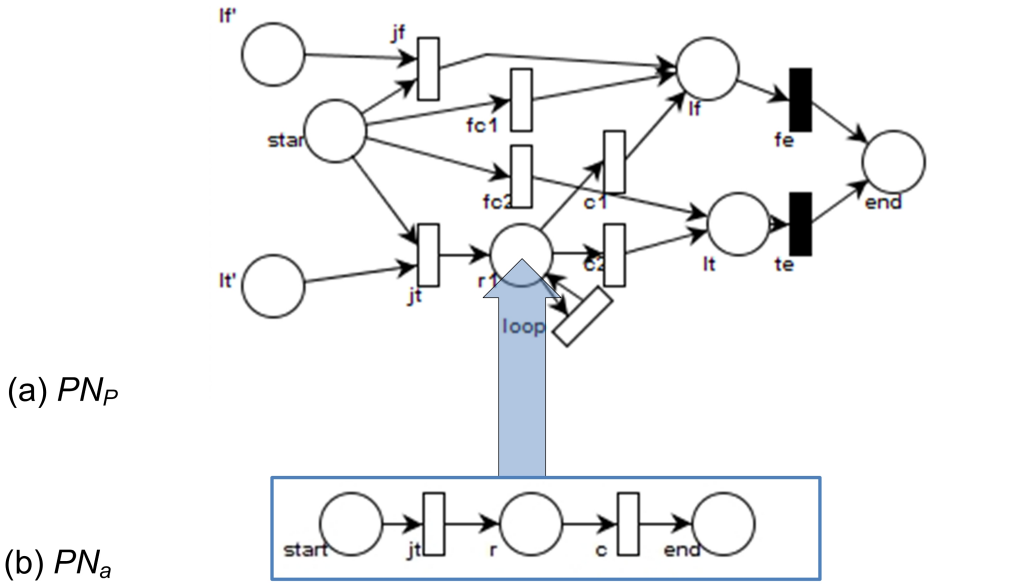
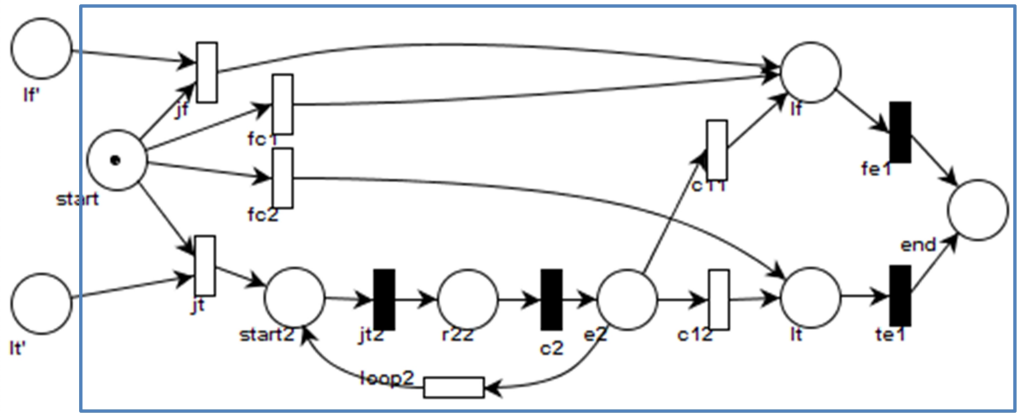


Figure 4.20 Hierarchical composition - *While*. (a) Control flow; (b) state-chart.



(a) PN_P

(b) PN_a



(c) PN_T

Figure 4.21 Hierarchical composition - *While*: Petri net.

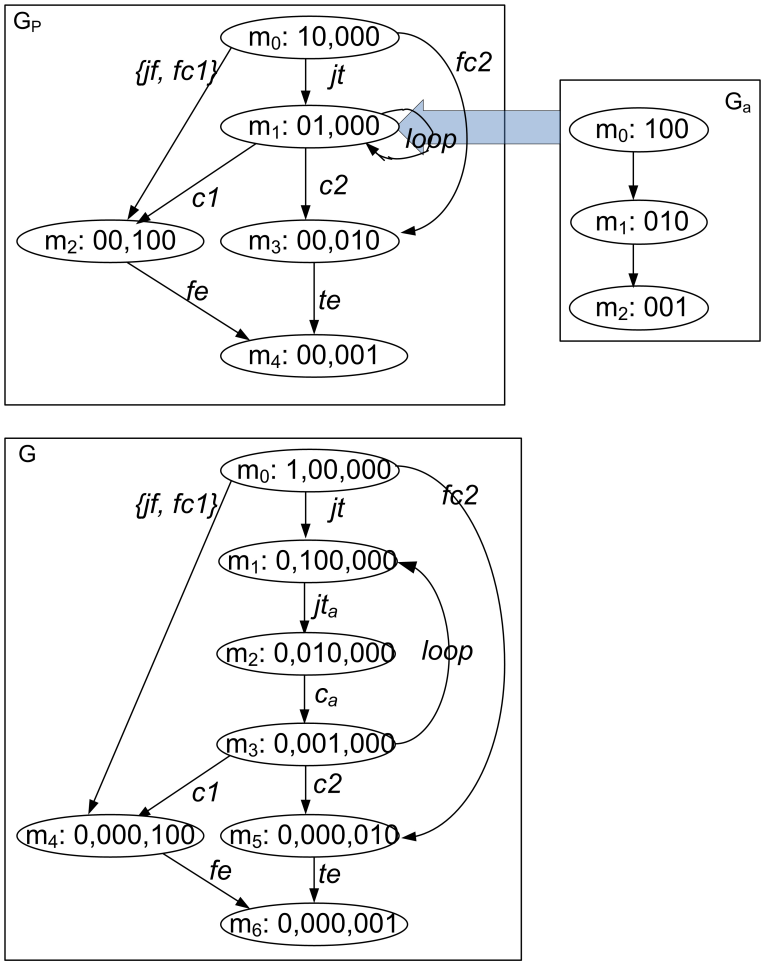


Figure 4.22 Hierarchical composition - *While*: Reachability graph.

The reachability graph of the composed Petri net PN_T, G shown in Figure 4.22, is isomorphic to the original statechart, as shown in Figure 4.20(b).

4.6.1.4 Correctness of atomic hierarchical composition with concurrency (No synchronization).

In this case, we consider concurrent composition without control links. As stated in section 4.1, state machines are used to describe the state transition of systems with concurrency.

The composition of all the structured activities follows the same "insertion" rule: replacing the running state r with the state transition of the component systems, denoted as S_a. However, S_a is different for activities with or without concurrency.

1. For activities without concurrency, the S_a is simply the state transition model of the single component activity.

2. *Flow* and parallel *ForEach* have multiple concurrent components. Therefore the S_a is the product of the component state transition models.

Let T be the structured activity that is composed using concurrent activities $\{T_i\}$, where $|T_i| = n$. PN_P is the Petri net generated for T according to its activity type during atomic transformation. PN_i is the Petri net modelling T_i.

$PN_P = (P_P, T_P, IN_P, OUT_P, R_P, M_{0P})$,

$PN_i = (P_i, T_i, IN_i, OUT_i, R_i, M_{0i})$, where $1 \leq i \leq n$

Then $PN_T = PN_P \bigodot \{PN_i\} = (P, T, IN, OUT, R, M_0)$, where

$P = P_P \bigcup \{P_i\} - \{r_i\}_P$

$T = T_P \bigcup \{T_i\}$

$IN = IN_P \bigcup \{IN_i\} \bigcup \{(\{end_i\}, c1), (\{end_i\}, c2)\}_P - \{(\{r_i\}, c1), (\{r_i\}, c2)\}_P$

$OUT = OUT_P \bigcup \{OUT_i\} \bigcup \{(jt, \{start_i\})\}_P - \{(jt, \{r_i\})\}_P$

$R = R_P \bigcup \{R_i\}$

$M_0 = \{[start_P]\}$.

PN_P, PN_i, PN_T are shown in Figure 4.23.

$G_P = (V_P, E_P)$ is the reachability graph of PN_P that is isomorphic to the statechart $S_P = (A_P, B_P)$.

$G_i = (V_i, E_i)$ is the reachability graph of PN_i.

Let G_a be the product of G_i, i.e., $G_a = G_1 \times \ldots \times G_n = (V_a, E_a)$.

$V_a = V_1 \times V_2 \times \ldots \times V_n$,

$E_a = \{(p_1 \ldots p_n, q_1 \ldots q_n) \mid \exists\ i$ such that $(p_i, q_i) \in E_i \wedge (\forall j, 1 \leq j \leq n$ and $j \neq i,$

$p_j = q_j)\}$

–$[start_a] = [start_1 \ldots start_i \ldots start_n]$

–$[end_a] = [end_1 \ldots end_i \ldots end_n]$.

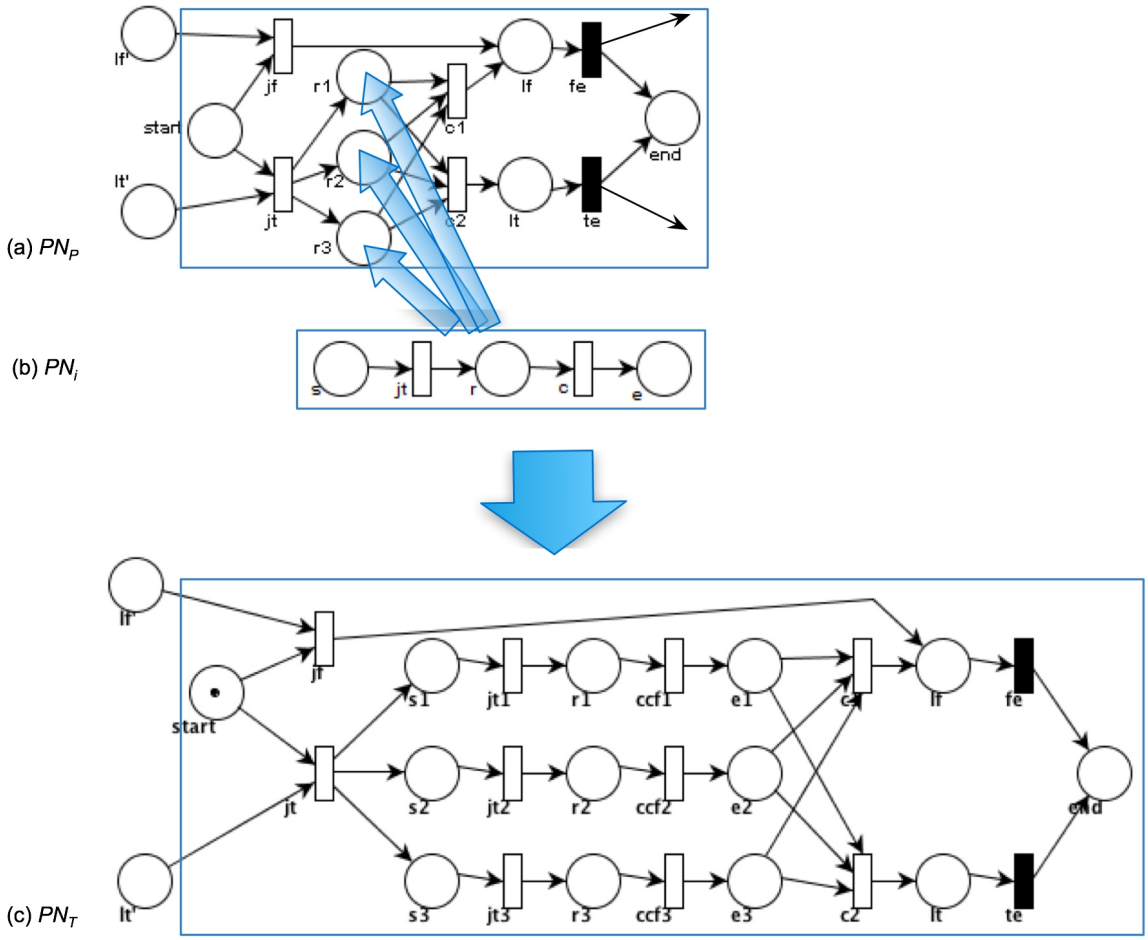


Figure 4.23 Hierarchical composition with concurrency.

Then we have $G = (V, E)$, where

$$V = V_P \bigcup V_a - \{[\{r_i\}_P]\}$$

$$E = E_P \bigcup E_a$$

$$\bigcup\{(s_P, [start_a]) \mid \exists s_P \in V_P, (s_P, [\{r_i\}_P]) \in E_P,$$

$$([end_a], t_P) \mid \exists t_P \in V_P, ([\{r_i\}_P], t_P) \in E_P\}$$

$$-\{(s_P, [\{r_i\}_P]) \mid \exists s_P \in V_P, (s_P, [\{r_i\}_P]) \in E_P,$$

$$([\{r_i\}_P], t_P) \mid \exists t_P \in V_P, ([\{r_i\}_P], t_P) \in E_P\}$$

Let $E_b = \{(s_P, [start_a]) \mid \exists s_P \in V_P, (s_P, [\{r_i\}_P]) \in E_P,$

$$([end_a], t_P) \mid \exists t_P \in V_P, ([\{r_i\}_P], t_P) \in E_P\}$$

$$E_{ps} = \{(s_P, [\{r_i\}_P]) \mid \exists s_P \in V_P, (s_P, [\{r_i\}_P]) \in E_P,$$

$$([\{r_i\}_P], t_P) \mid \exists t_P \in V_P, ([\{r_i\}_P], t_P) \in E_P\}$$

Then $E = (E_P - E_{ps}) \bigcup E_a \bigcup E_b$.

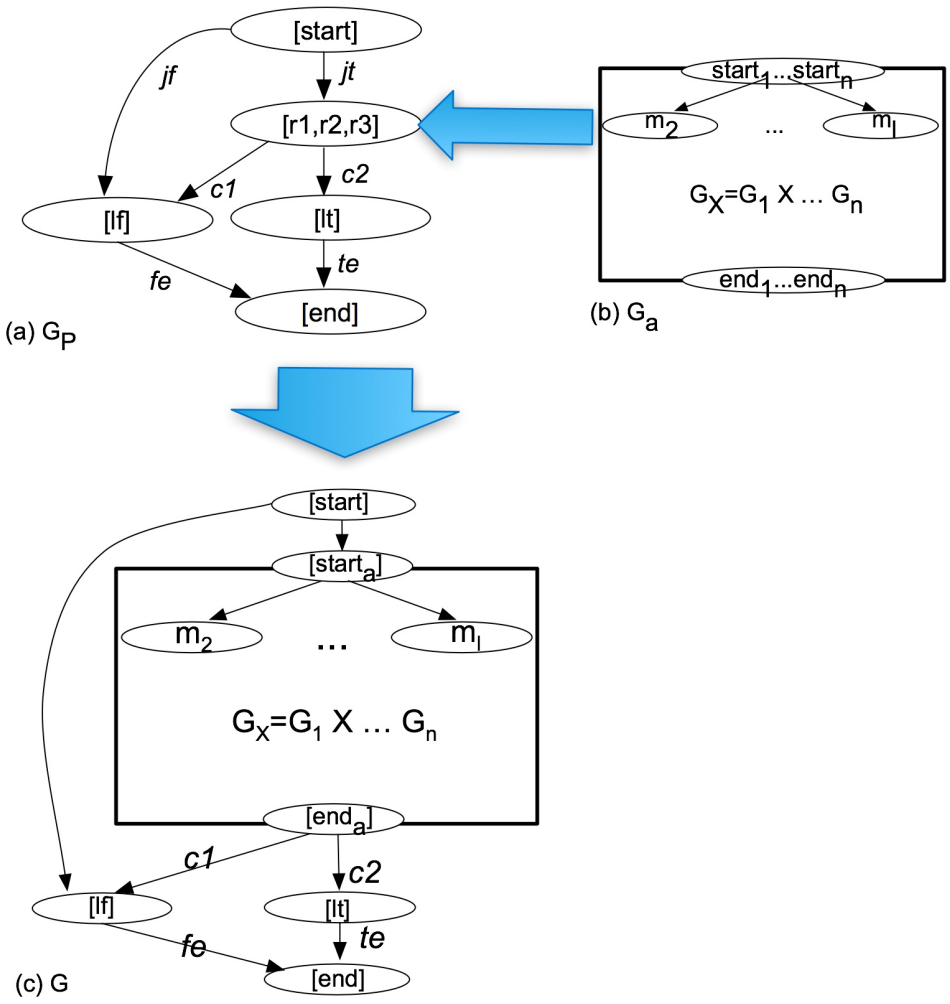


Figure 4.24 Hierarchical composition with concurrency.

The composition of G from G_P and G_a is shown in Figure 4.24.

Similarly, $S_i = (A_i, B_i)$ is the statechart of activity T_i.

Let S_a be the product of S_i, i.e., $S_a = S_1 \times S_2 \times \ldots \times S_n = (A_a, B_a)$, where

$A_a = A_1 \times A_2 \times \ldots \times A_n,$

$B_a = \{(k_1 \ldots k_n, r_1 \ldots r_n) \mid \exists\ i \text{ such that } (k_i, r_i) \in B_i \wedge (\forall j,\ 1 \leq j \leq n \text{ and } (j \neq i),$

$k_j = r_j)\}$

The statechart of T is $S = (A, B)$, where

$A = A_P \bigcup A_a - \{\{r_i\}_P\}$

$B = B_P \bigcup B_a$

$\bigcup\{(ss_P, start_a) \mid \exists ss_P \in A_P, (ss_P, \{r_i\}_P) \in B_P,$

$(end_a, tt_P) \mid \exists tt_P \in A_P, (\{r_i\}_P, tt_P) \in B_P\}$

$-\{(ss_P, \{r_i\}_P) \mid \exists ss_P \in A_P, (s_P, \{r_i\}_P) \in B_P,$

$(\{r_i\}_P, tt_P) \mid \exists tt_P \in A_P, (\{r_i\}_P, tt_P) \in B_P\}$

$= (B_P - B_{ps}) \bigcup B_a \bigcup B_b$

–$start_a = start_1 \ldots start_i \ldots start_n$

–$end_a = end_1 \ldots end_i \ldots end_n.$

Theorem 5. *Based on the definitions above, if $G_i \cong S_i$, then $G_a \cong S_a$.*

Proof. Since $G_i \cong S_i$, there exists a bijection function $f_i : V_i \to A_i$, such that

$$(x, y) \in E_i \Leftrightarrow (f_i(x), f_i(y)) \in B_i \tag{4.7}$$

Define a new bijection function $f : V_a \to A_a$ as

$$f(x) = f_1(x_1) ... f_n(x_n), \text{when } x = x_1 x_2 ... x_n \in V_a \tag{4.8}$$

Since $[start_i] \in V_i$ and $start_i \in A_i$, with 4.7 we have $f_i([start_i]) = start_i$.

With 4.7 and 4.8, we have

$f([start_a]) = f([start_1 ... start_n]) = f_1([start_1]) \ldots f_i([start_i]) \ldots f_n([start_n])$

$= start_1 \ldots start_i \ldots start_n = start_a.$

$f([end_a]) = end_a.$

To prove $G_a \cong S_a$, we need to prove that $(f(x), f(y)) \in B_a \Leftrightarrow (x, y) \in E_a$.

$(f(x), f(y)) \in B_a$

$\Leftrightarrow (f(x), f(y)) = (f_1(x_1)...f_n(x_n), f_1(y_1)...f_n(y_n)) \in B_a$

$\Leftrightarrow (f(x), f(y)) = (f_1(x_1)...f_n(x_n), f_1(y_1)...f_n(y_n)) \wedge \exists i$ such that

$(f_i(x_i), f_i(y_i)) \in B_i \wedge$

$\forall j (1 \leq j \leq n$ and $j \neq i$, $f_j(x_j) = f_j(y_j))$

$\Leftrightarrow (x, y) = (x_1...x_n, y_1...y_n) \wedge \exists i$ such that

$(x_i, y_i) \in E_i \wedge$ /*by equation (4.7)*/

$\forall j (1 \leq j \leq n$ and $j \neq i$, $x_j = y_j)$

$\Leftrightarrow (x, y) = (x_1...x_n, y_1...y_n) \wedge (x_1...x_n, y_1...y_n) \in E_a$

$\Leftrightarrow (x, y) \in E_a$ □

Theorem 6. *Based on the definitions of* G, S, G_a, S_a, G_P, S_P*, if* $G_P \cong S_P$ *and* $G_a \cong S_a$*, then* $G \cong S$.

Proof. The proof is the same with the one for Theorem 3. □

Theorem 6 proves that the *Flow* activity and parallel *ForEach* (*completionCondition* is not specified) have reachability graphs isomorphic to the original state transition systems.

4.6.1.5 Correctness of atomic hierarchical composition - Parallel *ForEach* with *completionCondition* specified.

When composing a parallel *ForEach* activity with *completionCondition*, our first step follows the same insertion rule: we insert the Petri nets of instances into the parent Petri net. After the first step, we add more transitions and marking-dependent arcs.

Revisiting section 4.4.7.2, when the *completionCondition* is specified, any instance completion can terminate all other running instances.

Let T be the *ForEach* structured activity that is composed by T_i. T_i can be atomic activity or structured activity. PN_i is the Petri net for instance T_i. The starting place and ending

place in PN_i are s_i and e_i, respectively. PN_P is the Petri net generated for T from atomic transformation. $PN_P = PN_{flow}$.

T_i does not have incoming or outgoing control links according to the properties of control links (section 4.3). Therefore, PN_i has at most two transitions immediately preceding the end place e_i. If T_i is a *While* activity or a *sequential ForEach* activity, then in PN_i there are two transitions pointing to e_i: fc and c. Transition fc represents the situation of *"while condition is not satisfied"* and transition c represents the completion of activity execution. If T_i is not a *While* or *sequential ForEach* activiy, PN_i has only one transition c pointing to e_i. For convenience, in the description of $PN_{par\_foreach}$ shown below, we use c_{ik} to represent a transition that points to e_i in PN_i, where $k \in \{1, 2\}$. In fact, $\{c_{ik}\} = \{fc_i, c_i\}$ when T_i is a *While* or a *sequential ForEach* activity. Otherwise $\{c_{ik}\} = \{c_i\}$.

The Petri net generated for this activity is:

$PN_T = (P, T, IN, OUT, D^-, R, M_0)$, where

$P = P_P \bigcup \{P_i\} - \{r_i\}_P$

$T = T_P \bigcup \{T_i \bigcup \{ccf_{ik}, cct_{ik} | \exists c_{ik}, (c_{ik}, e_i) \in OUT_i\}_k - \{c_{ik} | \exists c_{ik}, (c_{ik}, e_i) \in OUT_i\}_k\}_i$

$IN = (IN_P \bigcup \{(\{e_i\}, c1), (\{e_i\}, c2)\} - \{(\{r_i\}, c1), (\{r_i\}, c2)\}_P)$

$\bigcup \{IN_i \bigcup \{(p_i, cct_{ik}), (p_i, ccf_{ik}) | \exists p_i \in P_i, c_{ik} \in T_i, (p_i, c_{ik}) \in IN_i \text{ and } (c_{ik}, e_i) \in OUT_i\}_k$

$-\{(p_i, c_{ik}) | \exists p_i \in P_i, c_{ik} \in T_i, (p_i, c_{ik}) \in IN_i \text{ and } (c_{ik}, e_i) \in OUT_i\}_k\}_i$

$\bigcup \{(p_j, cct_{ik}) | \exists p_j \in P_j, c_{ik} \in T_i, i \neq j\}_{i,j}$

$OUT = (OUT_P \bigcup \{(jt, \{start_i\})\} - \{(jt, \{r_i\})\}_P)$

$\bigcup \{OUT_i \bigcup \{(ccf_{ik}, e_i), (cct_{ik}, e_i) | \exists c_{ik} \in T_i, (c_{ik}, e_i) \in OUT_i\}_k$

$-\{(c_{ik}, e_i) | \exists c_{ik} \in T_i, (c_{ik}, e_i) \in OUT_i\}_k\}_i$

$\bigcup \{(cct_{ik}, e_j) | \exists c_{ik} \in T_i, i \neq j\}_{i,j}$

$D^- = \{(\mu_{p_j}, (p_j, cct_{ik})) | \exists p_j \in P_j \text{ and } (p_j, cct_{ik}) \in IN, i \neq j\}_{k,i,j}$

$R = R_P \bigcup \{R_i \bigcup \{r_{ccf_{ik}}, r_{cct_{ik}} | \exists ccf_{ik}, cct_{ik} \in T\}_k - \{r_{c_{ik}} | \exists c_{ik} \in T_i \text{ and } (c_{ik}, e_i) \in OUT_i\}_k\}_i$

$M_0 = \{[start]\}$

- $1 \leq i \leq n, 1 \leq j \leq n$

- *n is the number of instances*

- μ_a *is the marking of place* a.

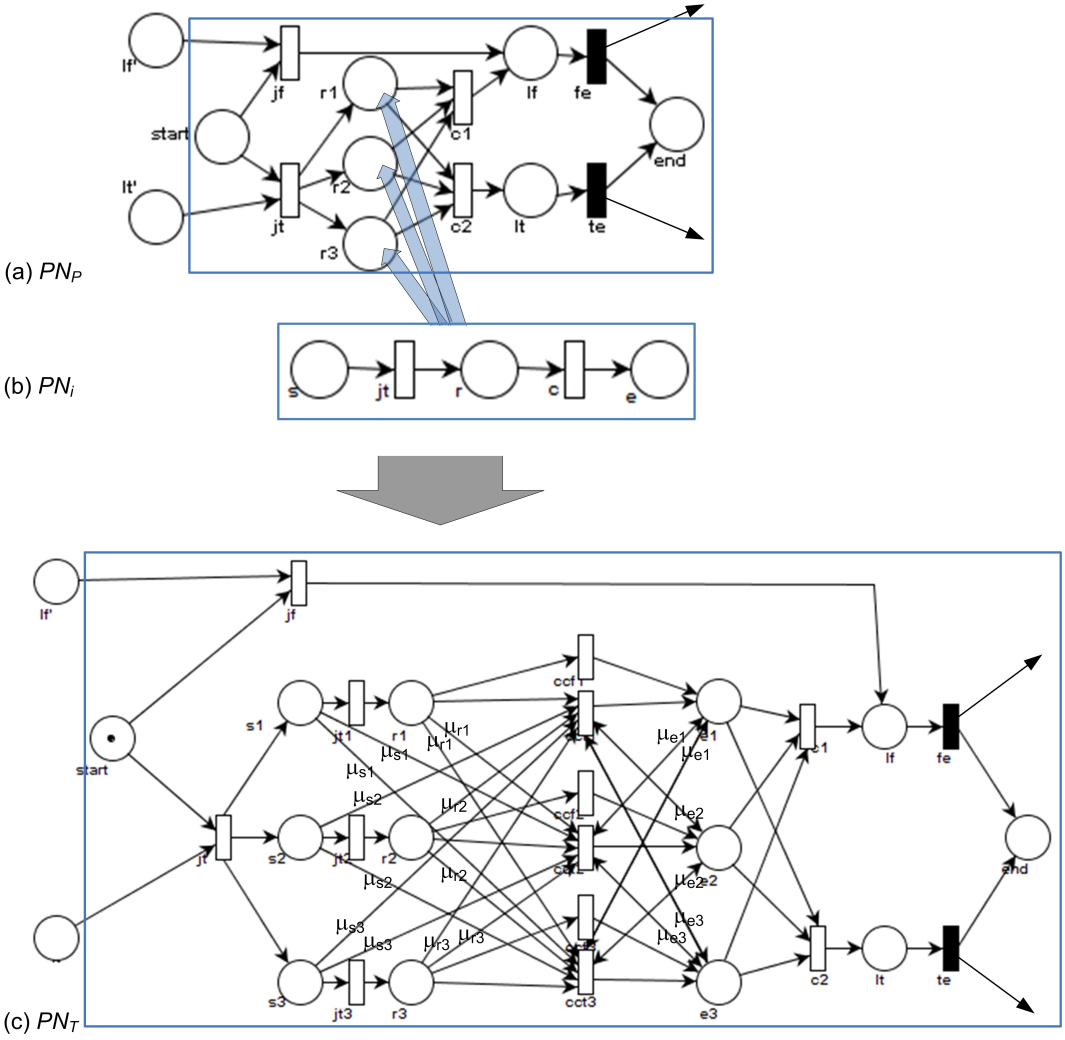


Figure 4.25 Parallel *ForEach* with *completionCondition*. (a) statechart; (b) Petri net.

The composition of PN_T is displayed in Figure 4.25. To ease the presentation, we set three instances of basic activity for this parallel $ForEach$ activity.

In this graph, we still insert the component Petri net PN_i into the the parent Petri net PN_P. After insertion, we replace each transition that points to the e_i place with a set of transitions: ccf_i and cct_i.

If the instances are "While" or "sequential ForEach" activities, then there are two sets of ccf_i *and* cct_i *transitions for each instance* i. This is because that there are two transitions

pointing to the end place in such activities (as discussed above).

All the places pertained to other instances are connected to the transition cct_i through marking-dependent arcs (18). The marking-dependent arc cardinalities are defined in D^-.

$\forall p_i \in P, \forall t_j \in T, D^-_{i,j} : IN^{|P|} \rightarrow IN$ is the marking-dependent cardinality of the input arc from p_i to t_j.

For example, the arc (s_i, cct_j) has cardinality μ_{s_i}, which is the marking of place s_i. When $\mu_{s_i} = 0$, this arc has no impact on transition cct_j, i.e., no token is removed from place s_i if cct_j fires. When $\mu_{s_i} = 1$, the token in place s_i is removed by the firing of cct_j.

This way, the firing of transition cct_j terminates all the instances in the Petri net.

When instance j is about to end (the execution finishes or the "while" condition is not satisfied), transitions ccf_j and cct_j are enabled at the same time. They stand for two conditions: "*completionCondition* is false" and "*completionCondition* is true", respectively. If the *completionCondition* is not satisfied, ccf_j fires and the token is moved from r_j to e_j. Instance j waits for other instances to complete. If the *completionCondition* is satisfied, transition cct_j is fired that removes all the tokens in this Petri net and put tokens in $\{e_i\}_i$, i.e., all other instances are terminated.

The state transition of the *ForEach* activity is shown in Figure Figure 4.26(a). The statechart displays the concurrent execution of the two instances. The statechart of each instance has three states: s_i, r_i, e_i, representing the states: start, running, and end. The r_i state may be hierarchically replaced using another component state machine G_{ai} if the instance is a structured activity. In this case, the curved arrow from r_i to e_i illustrates the transition from any state inside G_{ai} to e_i.

The transition from s_i to e_i and the transition from r_i (or G_{ai}) to e_i is enabled only when at least one other instance j is in state e_j and the *completionCondition* is true. The state transition for each instance is shown in Figure 4.26(b).

Lemma 1. *In Figure 4.25, $\{T_i\}$ is the set of concurrent instances, where $|T_i| = n$. S_i is the statechart of instance T_i, and G_i is the reachability graph of PN_i. $G_i \cong S_i$.*

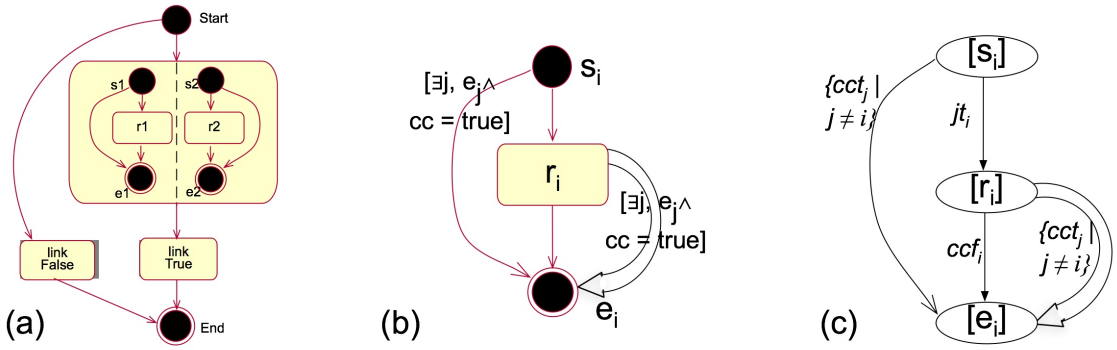


Figure 4.26 Instance state transition systems. (a) State machine of the *ForEach* activity; (b) state machine of each instance; (c) reachability graph of each instance.

Proof. Let $G'_i = (V'_i, E'_i)$ be the original reachability graph of PN_i.

Based on Figure 4.25, the new reachability graph of PN_i is $G_i = (V_i, E_i)$, where

$V_i = V'_i$.

$E_i = E'_i \bigcup \{(ss_i, [e_i]) \mid \exists ss_i \in V_i - \{[s_i], [e_i]\}, (ss_i, [e_i]) \notin E'_i,$

and $\exists j, i \neq j$, the current state in G_j is $[e_j]\wedge$ completionCondition is true) $\}$

$\bigcup \{([s_i], [e_i]) \mid ([s_i], [e_i]) \in E'_i$ and

("While" condition is not satisfied) or

$\exists j, i \neq j$, the current state in G_j is $[e_j]\wedge$ completionCondition is true)

or $(([s_i], [e_i]) \notin E'_i$ and

$\exists j, i \neq j$, the current state in G_j is $[e_j]\wedge$ completionCondition is true) $\}$

$-\{([s_i], [e_i]) \mid ([s_i], [e_i]) \in E'_i\}$

Similarly, the statechart of each instance is changed. Now each state in S_i, except r_i and e_i, can be directly transited to the e_i state when some other instance finishes and satisfies the "completionCondition". Let $S'_i = (A'_i, B'_i)$ be the original statechart of each instance. The new statechart S_i is:

$S_i = (A_i, B_i)$, where

$A_i = A'_i$.

$B_i = B_i' \bigcup \{(tt_i, e_i) \mid \exists tt_i \in A_i - \{s_i, e_i\}, (tt_i, e_i) \notin B_i',$

and $\exists j, i \neq j$, the current state in S_j is $e_j \wedge$ completionCondition is true) }

$\bigcup \{(s_i, e_i) \mid (s_i, e_i) \in B_i'$ and

("While" condition is not satisfied) or

$\exists j, i \neq j$, the current state in S_j is $e_j \wedge$ completionCondition is true)

or $((s_i, e_i) \notin B_i'$ and

$\exists j, i \neq j$, the current state in S_j is $e_j \wedge$ completionCondition is true) }

$- \{(s_i, e_i) \mid (s_i, e_i) \in B_i'\}$

The comparison of original statecharts and new statecharts for different types of activities is illustrated in Figure 4.27.

Since $G_i' \cong S_i'$, there exists a bijection function $f_i' : V_i' \rightarrow A_i'$ such that

$$(x, y) \in E_i' \Leftrightarrow (f_i'(x), f_i'(y)) \in B_i'$$

Now define a new bijection function $f_i = f_i'$.

$(f_i(x), f_i(y)) \in B_i$

$\Leftrightarrow$ (1)$(f_i'(x), f_i'(y)) \in B_i'$ and $(f_i'(x), f_i'(y)) \neq (s_i, e_i)$,

or (2) $((f_i'(x), f_i'(y)) \notin B_i') \wedge f_i'(x) \notin \{s_i, e_i\} \wedge f_i'(y) = e_i$

$\wedge \exists j, j \neq i$, the current state in S_j is $e_j \wedge$ completionCondition is true

or (3) $(f_i'(x), f_i'(y)) = (s_i, e_i)$ and

if $(f_i'(x), f_i'(y)) \in B_i'$, then (completionCondition is true or

$\exists j, i \neq j$, the current state in S_j is $e_j \wedge$ completionCondition is true)

else ($\exists j, i \neq j$, the current state in S_j is $e_j \wedge$ completionCondition is true)

$\Leftrightarrow$ (1)$(x, y) \in E_i'$ and $(x, y) \neq ([s_i], [e_i])$,

or (2) $((x, y) \notin E_i') \wedge x \notin \{[s_i], [e_i]\} \wedge y = [e_i]$

$\wedge \exists j, j \neq i$, the current state in G_j is $[e_j] \wedge$ completionCondition is true

or (3) $(x, y) = ([s_i], [e_i])$ and

if $(x, y) \in E_i'$, then (completionCondition is true or

$\exists j, i \neq j$, the current state in G_j is $[e_j] \wedge$ completionCondition is true)

else ($\exists j, i \neq j$, the current state in G_j is $[e_j]\wedge$ completionCondition is true)

$\Leftrightarrow (x, y) \in E_i$

Therefore, $G_i \cong S_i$. □

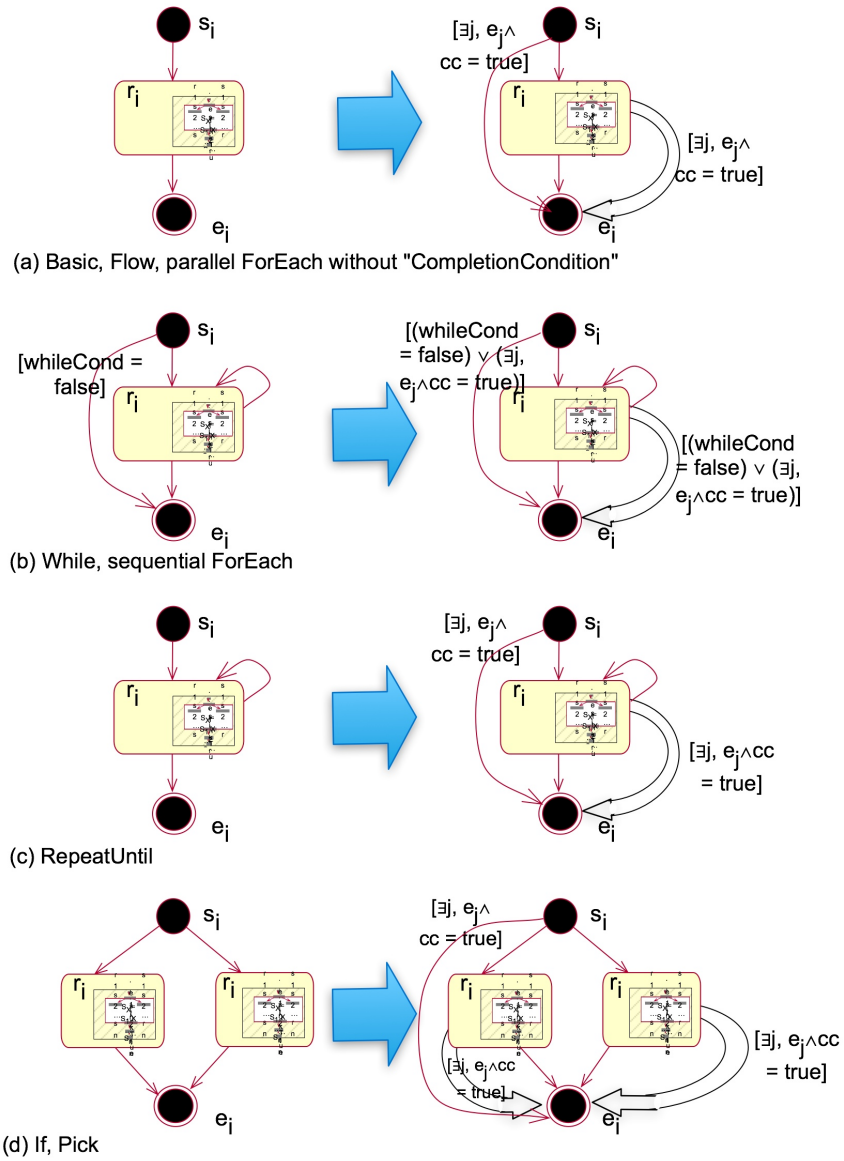


Figure 4.27 Original instance statecharts vs. Modified instance statecharts.

Let G_a be the product of G_i, i.e., $G_a = G_1 \times \ldots \times G_n = (V_a, E_a)$.

$V_a = V_1 \times ... \times V_n$

$E_a = \{(p_1...p_n, q_1...q_n)|$

$\exists i$ such that $(p_i, q_i) \in E_i \wedge \forall j(1 \leq j \leq n$ and $j \neq i$, $p_j = q_j)$

or

$\exists j, d, (1 \leq j, d \leq n, (p_j, q_j) \in E_j \wedge q_j = e_j, (p_d, q_d) \in E_d \wedge q_d = e_d)$

$\rightarrow \forall m, 1 \leq m \leq n, q_m = e_m\}$

$-[start_a] = [s_1 \ldots s_i \ldots s_n]$

$-[end_a] = [e_1 \ldots e_i \ldots e_n]$

Then we have $G = (V, E)$, where

$V = V_P \bigcup V_a - \{[\{r_i\}_P]\}$

$E = E_P \bigcup E_a$

$\bigcup\{(s_P, [start_a]) \mid \exists s_P \in V_P, (s_P, [\{r_i\}_P]) \in E_P,$

$([end_a], t_P) \mid \exists t_P \in V_P, ([\{r_i\}_P], t_P) \in E_P\}$

$-\{(s_P, [\{r_i\}_P]) \mid \exists s_P \in V_P, (s_P, [\{r_i\}_P]) \in E_P,$

$([\{r_i\}_P], t_P) \mid \exists t_P \in V_P, ([\{r_i\}_P], t_P) \in E_P\}$

Let $E_b = \{(s_P, [start_a]) \mid \exists s_P \in V_P, (s_P, [\{r_i\}_P]) \in E_P,$

$([end_a], t_P) \mid \exists t_P \in V_P, ([\{r_i\}_P], t_P) \in E_P\}$

$E_{ps} = \{(s_P, [\{r_i\}_P]) \mid \exists s_P \in V_P, (s_P, [\{r_i\}_P]) \in E_P,$

$([\{r_i\}_P], t_P) \mid \exists t_P \in V_P, ([\{r_i\}_P], t_P) \in E_P\}$

Then $E = (E_P - E_{ps}) \bigcup E_a \bigcup E_b$.

The composition of G from G_P and G_a is shown on the right of Figure 4.28.

Similarly, $S_i = (A_i, B_i)$ is the statechart of activity T_i.

Let S_a be the product of S_i, i.e., $S_a = S_1 \times S_2 \times \ldots \times S_n = (A_a, B_a)$, where

$A_a = A_1 \times ... \times A_n$

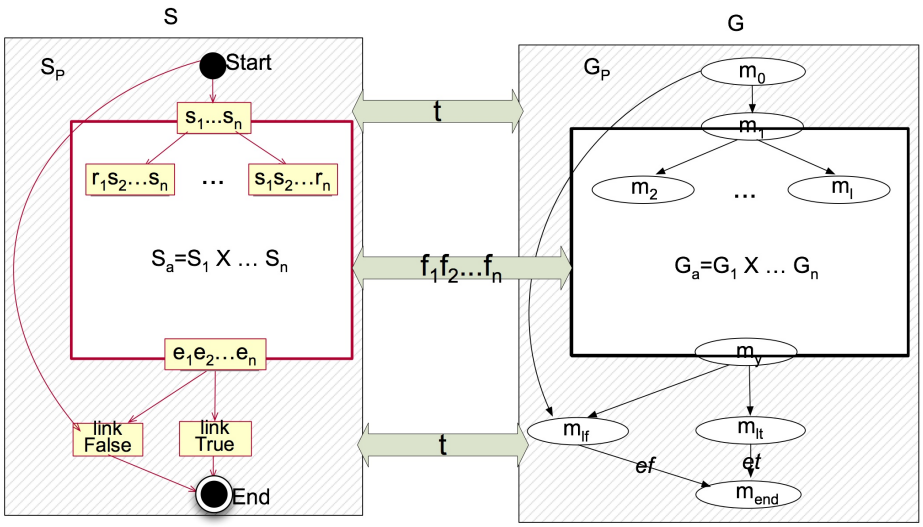


Figure 4.28 State transition systems for the parallel *ForEach* construct.

$$B_a = \{(k_1...k_n, r_1...r_n)|$$

$$\exists i \text{ such that } (k_i, r_i) \in B_i \wedge \forall j (1 \leq j \leq n \text{ and } j \neq i,\ k_j = r_j)$$

$$\text{or}$$

$$\exists j, d, (1 \leq j, d \leq n, (k_j, r_j) \in B_j \wedge r_j = e_j, (k_d, r_d) \in B_d \wedge r_d = e_d)$$

$$\rightarrow \forall m, 1 \leq m \leq n, r_m = e_m\}$$

The statechart of T is $S = (A, B)$, where

$$A = A_P \bigcup A_a - \{\{r_i\}_P\}$$

$$B = B_P \bigcup B_a$$

$$\bigcup\{(ss_P, start_a) \mid \exists ss_P \in A_P, (ss_P, \{r_i\}_P) \in B_P,$$

$$(end_a, tt_P) \mid \exists tt_P \in A_P, (\{r_i\}_P, tt_P) \in B_P\}$$

$$-\{(ss_P, \{r_i\}_P) \mid \exists ss_P \in A_P, (s_P, \{r_i\}_P) \in B_P,$$

$$(\{r_i\}_P, tt_P) \mid \exists tt_P \in A_P, (\{r_i\}_P, tt_P) \in B_P\}$$

$$= (B_P - B_{ps}) \bigcup B_a \bigcup B_b$$

$$-start_a = s_1 \ldots s_i \ldots s_n$$

$$-end_a = e_1 \ldots e_i \ldots e_n.$$

The composition of S from S_P and S_i is shown on the left of Figure 4.28

Lemma 2. *Based on the definitions of G_a, G_i, S_a, S_i, if $G_i \cong S_i$, then $G_a \cong S_a$.*

Proof. Since $G_i \cong S_i$, there exists a bijection function $f_i : V_i \to A_i$, such that

$$(x, y) \in E_i \Leftrightarrow (f_i(x), f_i(y)) \in B_i \tag{4.9}$$

Define a new bijection function $f : V_a \to A_a$ as

$$f(x) = f_1(x_1)...f_n(x_n), \text{when } x = x_1x_2...x_n \in V_a \tag{4.10}$$

Since $[s_i] \in V_i$ and $s_i \in A_i$, with 4.9 we have $f_i([s_i]) = s_i$.

With 4.9 and 4.10, we have

$$\begin{aligned} f([start_a]) &= f([s_1...s_n]) = f_1([s_1]) \dots f_i([s_i]) \dots f_n([s_n]) \\ &= s_1 \dots s_i \dots s_n = start_a. \end{aligned}$$

$$f([end_a]) = end_a.$$

To prove $G_a \cong S_a$, we need to prove that $(f(x), f(y)) \in B_a \Leftrightarrow (x, y) \in E_a$.

$(f(x), f(y)) \in B_a$

$\Leftrightarrow (f(x), f(y)) = (f_1(x_1)...f_n(x_n), f_1(y_1)...f_n(y_n)) \in B_a$ /*by equation (4.10)*/

$\Leftrightarrow (f(x), f(y)) = (f_1(x_1)...f_n(x_n), f_1(y_1)...f_n(y_n))$ and either of the following is satisfied:

(1) $\exists i$ such that $(f_i(x_i), f_i(y_i)) \in B_i \wedge \forall j (1 \le j \le n$ and $j \ne i$, $f_j(x_j) = f_j(y_j))$

(2) $\exists j, d, (1 \le j, d \le n, (f_j(x_j), f_j(y_j)) \in B_j \wedge f_j(y_j) = f_j(e_j), (f_d(x_d), f_d(y_d)) \in B_d$
$\wedge f_d(y_d) = f_d(e_d)) \rightarrow \forall m, 1 \le m \le n, f_m(y_m) = f_m(e_m)$

$\Leftrightarrow (x, y) = (x_1...x_n, y_1...y_n)$ and either of the following is satisfied:

(1) $\exists i$ such that $(x_i, y_i) \in E_i \wedge \forall j (1 \le j \le n$ and $j \ne i, x_j = y_j)$ /*by equation (4.9)*/

(2) $\exists j, d, (1 \le j, d \le n, (x_j, y_j) \in E_j \wedge y_j = e_j, (x_d, y_d) \in E_d$
$\wedge y_d = e_d) \rightarrow \forall m, 1 \le m \le n, y_m = e_m$

$\Leftrightarrow (x, y) = (x_1...x_n, y_1...y_n) \in E_X$

$\Leftrightarrow (x, y) \in E$

□

Theorem 3 demonstrates that when $G_a \cong S_a$ and $G_P \cong S_P$, we have $G \cong S$. Therefore, from Lemma 1, Lemma 2 and Theorem 3, we can derive the following theorem.

Theorem 7. *The composition of parallel ForEach activity preserves the original system behavior.*

4.6.1.6 Correctness of atomic hierarchical composition with Concurrency and Synchronization.

Control links are contained only in "Flow" activities to realize synchronization. They may change the original state transition of a business process. To analyze state transition with control links, we need to analyze the whole "Flow" activity that contains all the control links.

In section 4.3 we state that control links do not cross the boundaries of loops or parallel executions.

1. **Reachability graph G.**

   Firstly, we define the Petri net modelling the "Flow" activity with control links. Algorithm 5 describes how to add control links to the Petri net of "Flow" activity, PN_T. We depict the added arcs and places in Figure 4.29.

   In the $Flow$ activity, there are totally m join places (or join conditions). An arbitrary join condition d has z_d input links.

   As shown in Figure 4.29, the d'th join condition has z_d sets of $\{lf_{dk}, lt_{dk}\}_k$ places, where $1 \leq k \leq z_d$.

   The total number of added places in PN_T is $\sum_{d=1}^{m}(2 * z_d)$.

   According to the specification of WS-BPEL (81), each link has three status codes: $undecided, false$ and $true$. In Figure 4.29, the status of each link l_{dk} is represented using two places: lf_{dk} and lt_{dk}. The markings of $[lf_{dk}, lt_{dk}]$ have the following meanings:

$$l_{dk} = \begin{cases} true, & [lf_{dk}, lt_{dk}] = 01; \\ false, & [lf_{dk}, lt_{dk}] = 10; \\ undecided, & [lf_{dk}, lt_{dk}] = 00. \end{cases}$$

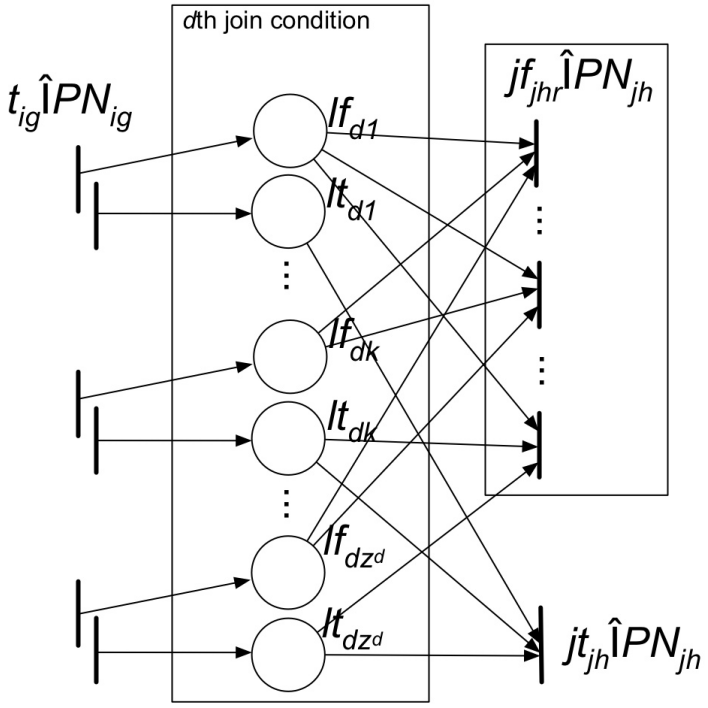


Figure 4.29 The d'th join condition in PN_T.

Note that $[lf_{dk}, lt_{dk}] = 11$ is not defined above because it is an infeasible marking in our generated Petri nets.

Let $V_{L_{ij}}$ be the set of possible states for link l_{ij}. We have

$$V_{L_{ij}} = \{true, false, undecided\}.$$

Furthermore, let V_{L_d} be the set of possible states for join condition d. We have

$V_{L_d} = \{$true $|\forall j, l_{dj} = true,$

$false\ |\exists k, l_{dk} = false \wedge \forall j, l_{dj} \neq undecided,$

$undecided\ |\exists k, l_{dk} = undecided\ \}.$

Let PN_P be the Petri net generated for T according to PN_{Flow} during atomic transformation. With PN_P, PN_i and the places/arcs added, we compose them into the Petri net PN_T:

$PN_T = (P, T, IN, OUT, R, M_0)$, where

$1 \le d \le m, 1 \le k \le z_d, 1 \le r \le (2^{z_d} - 1)$

$P = (P_P - \{r_{Pi}\}) \bigcup \{P_i\} \bigcup \{\{lt_{dk}\}, \{lf_{dk}\}\}_{k,d}$

$T = T_P \bigcup \{T_i\}$

$IN = IN_P \bigcup \{IN_i\} \bigcup \{(\{end_i\}, c1), (\{end_i\}, c2)\}_P - \{(\{r_i\}, c1), (\{r_i\}, c2)\}_P$

/* Added input arcs */

$\bigcup \{(\{lf_{dk} | lt_{dk}\}_k, \{jf_{jh}\}_r), (\{lt_{dk}\}_k, jt_{jh}),$

$\{(\{lt_{dk}\}_k, fc1_{jh}), (\{lt_{dk}\}_k, fc2_{jh}) | \exists fc1_{jh}, fc2_{jh} \in PN_{jh}\}$

$| \exists PN_{jh}$ that holds join condition $d\}_d$

$OUT = OUT_P \bigcup \{OUT_i\} \bigcup \{(jt, \{start_i\})\}_P - \{(jt, \{r_i\})\}_P$

/* Added output arcs */

$\bigcup \{(te_{ig}, lt_{dk}) | \exists T_{ig}$ is the source activity of link $l_{dk}\}_{k,d}$

$\bigcup \{(fe_{ig}, lf_{dk}) | \exists T_{ig}$ is the source activity of link l_{dk} and the transition

condition for l_{dk} is specified.$\}_{k,d}$

$R = R_P \bigcup \{R_i\}$

$M_0 = \{[start_P]\}$.

$G_P = (V_P, E_P)$.

$G_i = (V_i, E_i)$

Let G_a be the reachability graph of the concurrent execution part. We have $G_a = (V_a, E_a)$, where

$V_a = \{p_1 \dots p_n l_1 \dots l_m | \forall i, p_i \in V_i, \forall j, l_j \in V_{L_j}\}$

$E_a = \{(p_1 \dots p_n l_1 \dots l_m, q_1 \dots q_n l_1' \dots l_m') \mid$ either (1.1)-(1.3) or (2.1)-(2.4) or (3.1)-(3.4) are satisfied:

(1.1) $\exists i$ such that $(p_i, q_i) \in E_i$

(1.2) $\forall j, 1 \leq j \leq n \wedge j \neq i, (p_j = q_j)$

(1.3) $\forall k, l_k = l_k^{'}$

(1.4) $p_i \in V_{ig}$, T_{ig} is not associated with any control link, or

T_{ig} has incoming links $\wedge p_i \neq [start_{ig}]$, or

T_{ig} has outgoing links $\wedge r_i \neq [end_{ig}]$;

or

(2.1) $\exists i$ such that $(p_i, q_i) \in E_i$ and $q_i = [end_{ig}] \in V_{ig}$

(2.2) $\forall j, 1 \leq j \leq n \wedge j \neq i, (p_j = q_j)$

(2.3) $\exists \{(c, x)\}$, where $|\{(c, x)\}| = nl$, $1 \leq c \leq m, 1 \leq x \leq z_c$, such that

$/*$ $\{(c, x)\}$ represents the set of links $\{l_{cx}\}$ leaving T_{ig}. $*/$

$/*$ nl is the total number of T_{ig}'s output links. $*/$

$\exists t_{ig} \in T_i, (t_{ig}, end_{ig}) \in OUT_i \wedge (p_i, q_i)$ corresponds to the firing of t_{ig}

$\wedge (t_{ig}, lt_{cx}) \in OUT \wedge [lt_{cx}] = 0 \wedge [lt_{cx}]' = 1$

$/*$ $l_{cx} = undecided$ and $l'_{cx} = true$ $*/$

or

$\exists t_{ig} \in T_i, (t_{ig}, end_{ig}) \in OUT_i \wedge (p_i, q_i)$ corresponds to the firing of t_{ig}

$\wedge (t_{ig}, lf_{cx}) \in OUT \wedge [lf_{cx}] = 0 \wedge [lf_{cx}]' = 1$

$/*$ $l_{cx} = undecided$ and $l'_{cx} = false$ $*/$

(2.4) $\forall$ links l_{dy} and $l_{dy} \notin \{l_{cx}\}, l_{dy} = l_{dy}^{'}$;

or

(3.1) $\exists i$ such that $(p_i, q_i) \in E_i$ and $p_i = [start_{ig}]$

(3.2) $\forall j, 1 \leq j \leq n \wedge j \neq i, (p_j = q_j)$

(3.3) $\exists$ join condition l_c, such that

$/* \ T_{ig}$ holds join condition $l_c \ */$

$\exists t_{ig} \in T_i, (start_{ig}, t_{ig}) \in IN_i \wedge (p_i, q_i)$ corresponds to the firing of t_{ig}, and

(3.3.1) $q_i \notin \{[end_{ig}], [lf_{ig}] | lf_{ig} \in P_{ig}\} \wedge \forall x, (lt_{cx}, t_{ig}) \in IN \wedge l_c = true$

$\wedge l'_c = undecided$ $/*$ join true $*/$

or

(3.3.2) $q_i \in \{[end_{ig}], [lf_{ig}] | lf_{ig} \in P_{ig}\} \wedge \exists x, (lf_{cx}, t_{ig}) \in IN \wedge l_c = false$

$\wedge l'_c = undecided$ $/*$ join false $*/$

or

(3.3.3) $q_i = [end] \wedge l_c \neq undecided \wedge l'_c = undecided$

$/* \ T_{ig}$ is an "Exit" activity$*/$

(3.4) $\forall d \neq c$, join condition $l_d = l'_d$.

–$[start_a] = [start_i \dots start_n]$.

–$[end_a] = [end_i \dots end_n]$.

–Note that both $[start_a]$ and $[end_a]$ do not contain any token in places added for control links because all those added places are empty (all the links are undecided) when the "Flow" activity just starts or terminates.

Then we have the reachability graph of PN_T, $G = (V, E)$, where

$V = V_P \bigcup V_a - \{[\{r_i\}_P]\}$

$E = E_P \bigcup E_a$

$\bigcup \{(s_P, [start_a]) \mid \exists s_P \in V_P, (s_P, [\{r_i\}_P]) \in E_P,$

$([end_a], t_P) \mid \exists t_P \in V_P, ([\{r_i\}_P], t_P) \in E_P\}$

$-\{(s_P, [\{r_i\}_P]) \mid \exists s_P \in V_P, (s_P, [\{r_i\}_P]) \in E_P,$

$([\{r_i\}_P], t_P) \mid \exists t_P \in V_P, ([\{r_i\}_P], t_P) \in E_P\}.$

2. **State machine S.**

Suppose there is a link l from activity $A11$ to $A22$, where $A11$ and $A22$ belong to the concurrent paths $p1$ and $p2$, respectively. We only know the status of link l after $A11$ finishes. Let $\{A1_i\}$ be the set of activities in $p1$ that are executed after $A11$ finishes but before $A22$ starts. $\{A1_i\}$ have to carry the status of link l so that when $A22$ starts, the system knows whether to execute $A22$ or not. In other words, link status is part of the system state.

Let $A_{L_{ij}}$ be the set of possible states for link l_{ij}. We have

$A_{L_{ij}} = \{true, false, undecided\}$.

Let A_{L_d} be the set of possible states for join condition d. We have

$A_{L_d} = \{$true $|\forall j, l_{dj} = true,$

$false\ |\exists k, l_{dk} = false \wedge \forall j, l_{dj} \neq undecided,$

$undecided\ |\exists k, l_{dk} = undecided\ \}$,

where $l_{dj} = \{undecided, false, true\}$.

Let $S_i = (A_i, B_i)$ be the original state machine of component activity T_i.

Let S_a be the state transition of the concurrent execution of "Flow" activity T. We have:

$S_a = (A_a, B_a)$.

$A_a = \{k_1 \ldots k_n ll_1 \ldots ll_m | \forall i, k_i \in A_i, \forall j, ll_j \in A_{L_j}\}$,

$B_a = \{(k_1 \ldots k_n ll_1 \ldots ll_m,\ r_1 \ldots r_n ll'_1 \ldots ll'_m)$ | either (1.1)-(1.3) or (2.1)-(2.4) or (3.1)-(3.4) are satisfied:

(1.1) $\exists i, (k_i, r_i) \in B_i$

(1.2) $\forall j, 1 \leq j \leq n \wedge j \neq i, (k_j = r_j)$

(1.3) $\forall i, ll_i = ll'_i$

(1.4) $k_i \in A_{ig}$, T_{ig} is not associated with any control link, or

T_{ig} has incoming links $\wedge k_i \neq start_{ig}$, or

T_{ig} has outgoing links $\wedge r_i \neq end_{ig}$;

or

(2.1) $\exists i$ such that $(k_i, r_i) \in B_i$ and $r_i = end_{ig}$

(2.2) $\forall j, 1 \leq j \leq n \wedge j \neq i, (k_j = r_j)$

(2.3) k_i contains the status of links $\{l_{cx}\}_{|nl|}$, where nl is the number of output links for T_{ig}, such that:

$k_i = LinkTrue_{ig_{cx}} \wedge l_{cx} = undecided \wedge l'_{cx} = true$, or

$k_i = LinkFalse_{ig_{cx}} \wedge l_{cx} = undecided \wedge l'_{cx} = false$

(2.4) $\forall$ links l_{dy} and $ll_{dy} \notin \{l_{cx}\}, l_{dy} = l'_{dy}$;

or

(3.1) $\exists i$ such that $(k_i, r_i) \in B_i$ and $k_i = start_{ig}$

(3.2) $\forall j, 1 \leq j \leq n \wedge j \neq i, (k_j = r_j)$

(3.3) $\exists$ join condition l_c, T_{ig} holds join condition l_c, such that

$/* \ (k_i, r_i)$ represents join true $*/$

$r_i \neq (end_{ig} | LinkFalse_{ig}) \wedge l_c = true \wedge l'_c = undecided$, or

$/* \ (k_i, r_i)$ represents join false $*/$

$r_i = (end_{ig} | LinkFalse_{ig}) \wedge l_c = false \wedge l'_c = undecided$, or

$/* \ (k_i, r_i)$ represents the start of an "Exit" activity $*/$

$r_i = end \wedge l_c \neq undecided \wedge l'_c = undecided$

(3.4) $\forall d \neq c$, join condition $l_d = l'_d$.

–$start_a = start_i \ldots start_n \{undecided\}_{|m|}$

–$end_a = end_i \ldots end_n \{undecided\}_{|m|}$.

The statechart of T is $S = (A, B)$, where

$A = A_P \bigcup A_a - \{\{r_i\}_P\}$

$B = B_P \bigcup B_a$

$\bigcup\{(ss_P, start_a) \mid \exists ss_P \in A_P, (ss_P, \{r_i\}_P) \in B_P,$

$(end_a, tt_P) \mid \exists tt_P \in A_P, (\{r_i\}_P, tt_P) \in B_P\}$

$-\{(ss_P, \{r_i\}_P) \mid \exists ss_P \in A_P, (s_P, \{r_i\}_P) \in B_P,$

$(\{r_i\}_P, tt_P) \mid \exists tt_P \in A_P, (\{r_i\}_P, tt_P) \in B_P\}.$

3. **Isomorphism.**

Theorem 8. *Based on the definitions above, if $G_i \cong S_i$, then $S_a \cong G_a$.*

Proof. Define a bijection function $f_{ij} : V_{L_{ij}} \Rightarrow A_{L_{ij}}$ such that

$$f_{ij}(l_{ij}) = ll_{ij}, \text{when } l_{ij} \in V_{L_{ij}} \tag{4.11}$$

Define a bijection function $f_{L_i} : V_{L_i} \Rightarrow A_{L_i}$ such that

$$f_{L_i}(l_i) = ll_i, \text{when } l_i \in V_{L_i} \tag{4.12}$$

Define a bijection function $f : V_a \rightarrow A_a$ as

$$f(x) = f_1(x_1)...f_n(x_n)f_{L_1}(l_1)...f_{L_m}(l_m), \text{ if } x = x_1x_2...x_nl_1...l_m \in V_a.$$

To prove that $G_a \cong S_a$, we need to prove that $(f(x), f(y)) \in B_a \Leftrightarrow (x, y) \in E_a$.

$(f(x), f(y)) \in B_a$

$\Leftrightarrow (f_1(x_1)...f_n(x_n)f_{L_1}(l_1)...f_{L_m}(l_m), f_1(y_1)...f_n(y_n)f_{L_1}(l_1)'...f_{L_m}(l_m)') \in B_a$

$\Leftrightarrow$ Either (1.1)-(1.3) or (2.1)-(2.4) or (3.1)-(3.4) are satisfied:

(1.1) $\exists i, (f_i(x_i), f_i(y_i)) \in B_i$

(1.2) $\forall j, 1 \leq j \leq n \wedge j \neq i, f_j(x_j) = f_j(y_j)$

(1.3) $\forall i, f_{L_i}(l_i) = f_{L_i}(l_i)'$

(1.4) $f_i(x_i) \in A_{ig}$, T_{ig} is not associated with any control link, or

T_{ig} has incoming links $\wedge f_i(x_i) \neq start_{ig}$, or

T_{ig} has outgoing links $\wedge f_i(y_i) \neq end_{ig}$;

or

(2.1) $\exists i$ such that $(f_i(x_i), f_i(y_i)) \in B_i$ and $f_i(y_i) = end_{ig}$

(2.2) $\forall j, 1 \leq j \leq n \wedge j \neq i, f_j(x_j) = f_j(y_j)$

(2.3) $f_i(x_i)$ contains the status of links $\{f_{L_{cx}}(l_{cx})\}_{|nl|}$, where nl is the number of output links for T_{ig}, such that:

$f_i(x_i) = LinkTrue_{ig_{cx}} \wedge f_{L_{cx}}(l_{cx}) = undecided \wedge f_{L_{cx}}(l_{cx})' = true$, or

$f_i(x_i) = LinkFalse_{ig_{cx}} \wedge f_{L_{cx}}(l_{cx}) = undecided \wedge f_{L_{cx}}(l_{cx})' = false$

(2.4) $\forall$ links $f_{L_{dy}}(l_{dy})$ and $f_{L_{dy}}(l_{dy}) \notin \{f_{L_{cx}}(l_{cx})\}, f_{L_{dy}}(l_{dy}) = f_{L_{dy}}(l_{dy})'$;

or

(3.1) $\exists i$ such that $(f_i(x_i), f_i(y_i)) \in B_i$ and $f_i(x_i) = start_{ig}$

(3.2) $\forall j, 1 \leq j \leq n \wedge j \neq i, f_j(x_j) = f_j(y_j)$

(3.3) $\exists$ join condition $f_{L_c}(l_c)$, T_{ig} holds join condition $f_{L_c}(l_c)$, such that

$/*$ $(f_i(x_i), f_i(y_i))$ requires true join condition $*/$

$f_i(y_i) \neq (end_{ig}|LinkFalse_{ig}) \wedge f_{L_c}(l_c) = true \wedge f_{L_c}(l_c)' = undecided$, or

$/*$ $(f_i(x_i), f_i(y_i))$ requires false join condition $*/$

$f_i(y_i) = (end_{ig}|LinkFalse_{ig}) \wedge f_{L_c}(l_c) = false \wedge f_{L_c}(l_c)' = undecided$, or

$/*$ $(f_i(x_i), f_i(y_i))$ represents the start of an "Exit" activity $*/$

$f_i(y_i) = end \wedge f_{L_c}(l_c) \neq undecided \wedge f_{L_c}(l_c)' = undecided$

(3.4) $\forall d \neq c$, join condition $f_{L_d}(l_d) = f_{L_d}(l_d)'$.

$\Leftrightarrow$ Either (1.1)-(1.3) or (2.1)-(2.4) or (3.1)-(3.4) are satisfied:

(1.1) $\exists i, (x_i, y_i) \in V_i$

(1.2) $\forall j, 1 \leq j \leq n \wedge j \neq i, x_j = y_j$

(1.3) $\forall i, l_i = l'_i$

(1.4) $x_i \in V_{ig}$, T_{ig} is not associated with any control link, or

T_{ig} has incoming links $\wedge x_i \neq [start_{ig}]$, or

T_{ig} has outgoing links $\wedge y_i \neq [end_{ig}]$;

or

(2.1) $\exists i$ such that $(x_i, y_i) \in V_i$ and $y_i = [end_{ig}]$

(2.2) $\forall j, 1 \leq j \leq n \wedge j \neq i, x_j = y_j$

(2.3) x_i contains the status of links $\{l_{cx}\}_{|nl|}$, where $|\{(c, x)\}| = nl$, $1 \leq c \leq m, 1 \leq x \leq z_c$, such that

/*In other words, in OUT, the transition (x_i, y_i) corresponds to lt_{cx} or lf_{cx} */

$[lt_{cx}] = 0 \wedge [lt_{cx}]' = 1$, or

$[lf_{cx}] = 0 \wedge [lf_{cx}]' = 1$

(2.4) $\forall$ links l_{dy} and $l_{dy} \notin \{l_{cx}\}, l_{dy} = l'_{dy}$;

or

(3.1) $\exists i$ such that $(x_i, y_i) \in V_i$ and $x_i = [start_{ig}]$

(3.2) $\forall j, 1 \leq j \leq n \wedge j \neq i, x_j = y_j$

(3.3) $\exists$ join condition l_c, T_{ig} holds join condition l_c, such that

$y_i \notin \{[end_{ig}], [lf_{ig}] | lf_{ig} \in P_{ig}\} \wedge l_c = true \wedge l'_c = undecided$, or

$y_i \in \{[end_{ig}], [lf_{ig}] | lf_{ig} \in P_{ig}\} \wedge l_c = false \wedge l'_c = undecided$, or

$y_i = [end] \wedge l_c \neq undecided \wedge l'_c = undecided$

(3.4) $\forall d \neq c$, join condition $l_d = l'_d$

$\Leftrightarrow$ Either (1.1)-(1.3) or (2.1)-(2.4) or (3.1)-(3.4) are satisfied:

(1.1) $\exists i, (x_i, y_i) \in V_i$

(1.2) $\forall j, 1 \leq j \leq n \wedge j \neq i, x_j = y_j$

(1.3) $\forall i, l_i = l'_i$

(1.4) $x_i \in V_{ig}$, T_{ig} is not associated with any control link, or

T_{ig} has incoming links $\wedge x_i \neq [start_{ig}]$, or

T_{ig} has outgoing links $\wedge y_i \neq [end_{ig}]$;

or

(2.1) $\exists i$ such that $(x_i, y_i) \in V_i$ and $y_i = [end_{ig}]$

(2.2) $\forall j, 1 \leq j \leq n \wedge j \neq i, x_j = y_j$

(2.3) $\exists \{(c, x)\}$, where $|\{(c, x)\}| = nl$, $1 \leq c \leq m, 1 \leq x \leq z_c$, such that

$\exists t_{ig} \in T_i, (t_{ig}, end_{ig}) \in OUT_i \wedge (x_i, y_i)$ corresponds to the firing of t_{ig}

$\wedge (t_{ig}, lt_{cx}) \in OUT \wedge [lt_{cx}] = 0 \wedge [lt_{cx}]' = 1$

$/* \; l_{cx} = undecided$ and $l'_{cx} = true \; */$

or

$\exists t_{ig} \in T_i, (t_{ig}, end_{ig}) \in OUT_i \wedge (x_i, y_i)$ corresponds to the firing of t_{ig}

$\wedge (t_{ig}, lf_{cx}) \in OUT \wedge [lf_{cx}] = 0 \wedge [lf_{cx}]' = 1$

$/* \; l_{cx} = undecided$ and $l'_{cx} = false \; */$

(2.4) $\forall$ links l_{dy} and $l_{dy} \notin \{l_{cx}\}, l_{dy} = l'_{dy}$;

or

(3.1) $\exists i$ such that $(x_i, y_i) \in V_i$ and $x_i = [start_{ig}]$

(3.2) $\forall j, 1 \leq j \leq n \wedge j \neq i, x_j = y_j$

(3.3)$\exists$ join condition l_c, such that

$/* \; T_{ig}$ holds join condition $l_c \; */$

$\exists t_{ig} \in T_i, (start_{ig}, t_{ig}) \in IN_i \wedge (x_i, y_i)$ corresponds to the firing of t_{ig}, and

$/*$ join true $*/$

$y_i \notin \{[end_{ig}], [lf_{ig}] | lf_{ig} \in P_{ig}\} \wedge \forall x, (lt_{cx}, t_{ig}) \in IN \wedge l_c = true \wedge l'_c = undecided$

or

/* join false */

$y_i \in \{[end_{ig}], [lf_{ig}] | lf_{ig} \in P_{ig}\} \wedge \exists x, (lf_{cx}, t_{ig}) \in IN \wedge l_c = false \wedge l'_c = undecided$

or

$y_i = [end] \wedge l_c \neq undecided \wedge l'_c = undecided$

(3.4) $\forall d \neq c$, join condition $l_d = l'_d$

$\Leftrightarrow (x, y) \in E_a$.

Therefore, $S_a \cong G_a$.

□

Theorem 3 demonstrates that when $G_a \cong S_a$ and $G_P \cong S_P$, we have $G \cong S$. Therefore, from Theorem 8 and Theorem 3, we can derive the following theorem.

Theorem 9. *The composition of Flow activity with control links preserves the original system behavior.*

Example.

In Figure 4.30, we study a "Flow" activity that has two parallel child sequences. Each sequence has two basic activities. To ease the presentation, there is only one control link going from the *Act11* activity to the *Act22* activity (the control link is not shown in the control flow Figure 4.30(a)). That means activity *Act22* waits for *Act11* to finish and to pass link status. If the link status is "true" then activity *Act22* executes, otherwise activity *Act22* skips execution and terminates.

As described in Algorithm 1, we process control links after transforming all the component activities for *Flow*. The processing of control links follows Algorithm 5. In this example, we have only one link. For this link, we create two extra places: *lf'* and *lt'*. They represent the

join condition for *Act22*. As pointed out in section 4.4.1.1, when *Act22* has multiple incoming links, the place *lt'* will be duplicated for each incoming links, as depicted in Figure 4.1. *Act22* has only one *lf'* place to represent the *join false* condition even if it has multiple incoming links. This way, any false incoming link can enable the *jf* transition that skips the activity execution.

Note that when any incoming link has "false" status (place *lf'* has one token), its corresponding *lt'* place does not hold tokens. At this time, transition *jt* is not enabled. In other words, for one activity, its *jf* and *jt* will not be enabled at the same time.

The state transition systems of the original control flow and the Petri net are depicted in Figure 4.31. In Figure 4.31(a), S_{11}, A_{11} and E_{11} represent the starting, running, and ending states of activity $Act11$. When $Act11$ is finished (the status of the link is known) and $Act22$ has not entered running state, we need to carry the link status information in the system states. Therefore, we use $\bar{S}_{12}$ to represent that "*Act12 is in start state and Act11 finished with a false link*". Similarly, $\bar{A}_{12}$ means "*Act12 is running and Act11 finished with a false link*".

The graphs shown in Figure 4.31 are isomorphic that means the composed "Flow" Petri net maintains the fidelity of the original business process behavior.

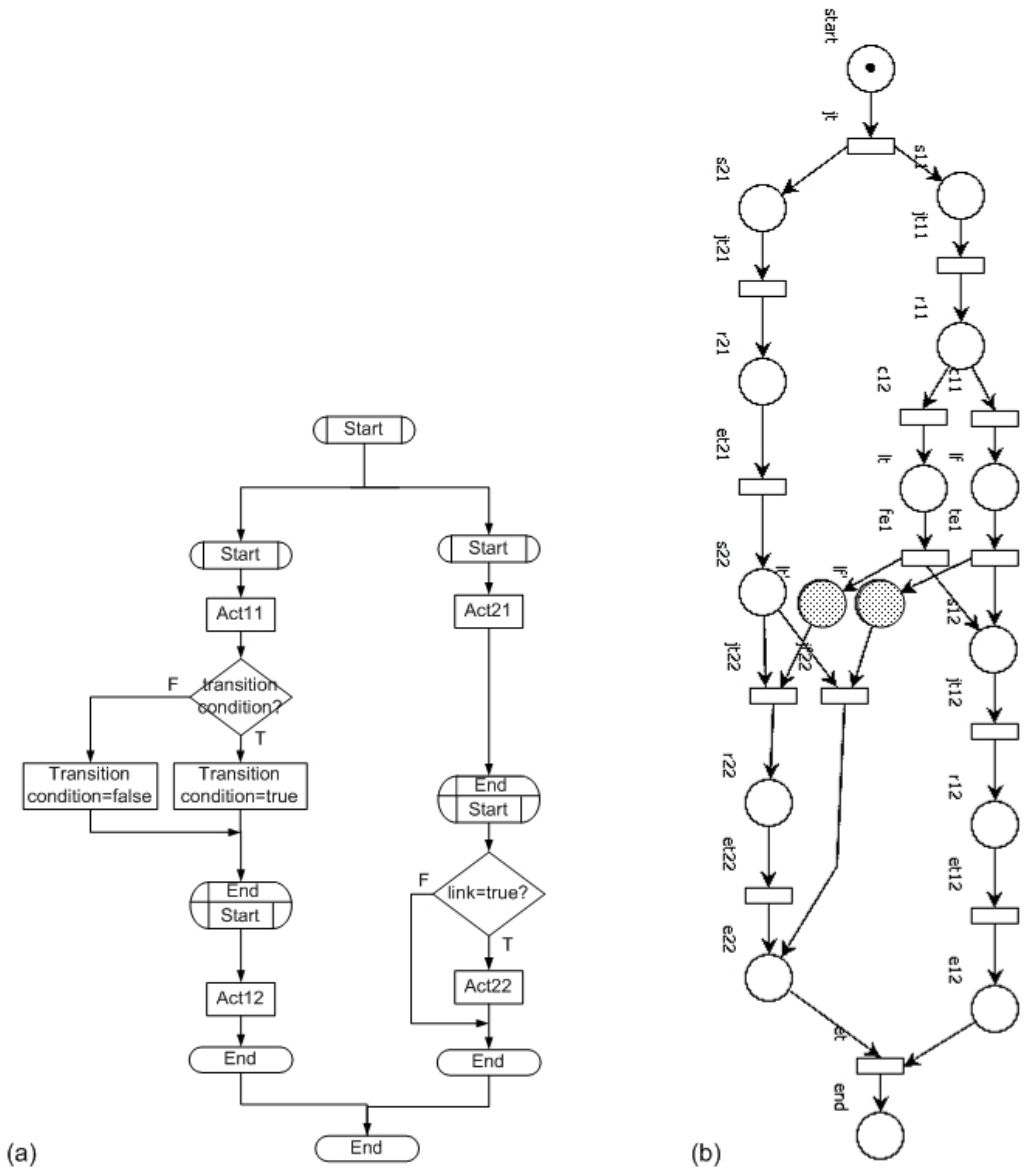


Figure 4.30 Hierarchical composition - *Flow*. (a) Control flow; (b) Petri net.

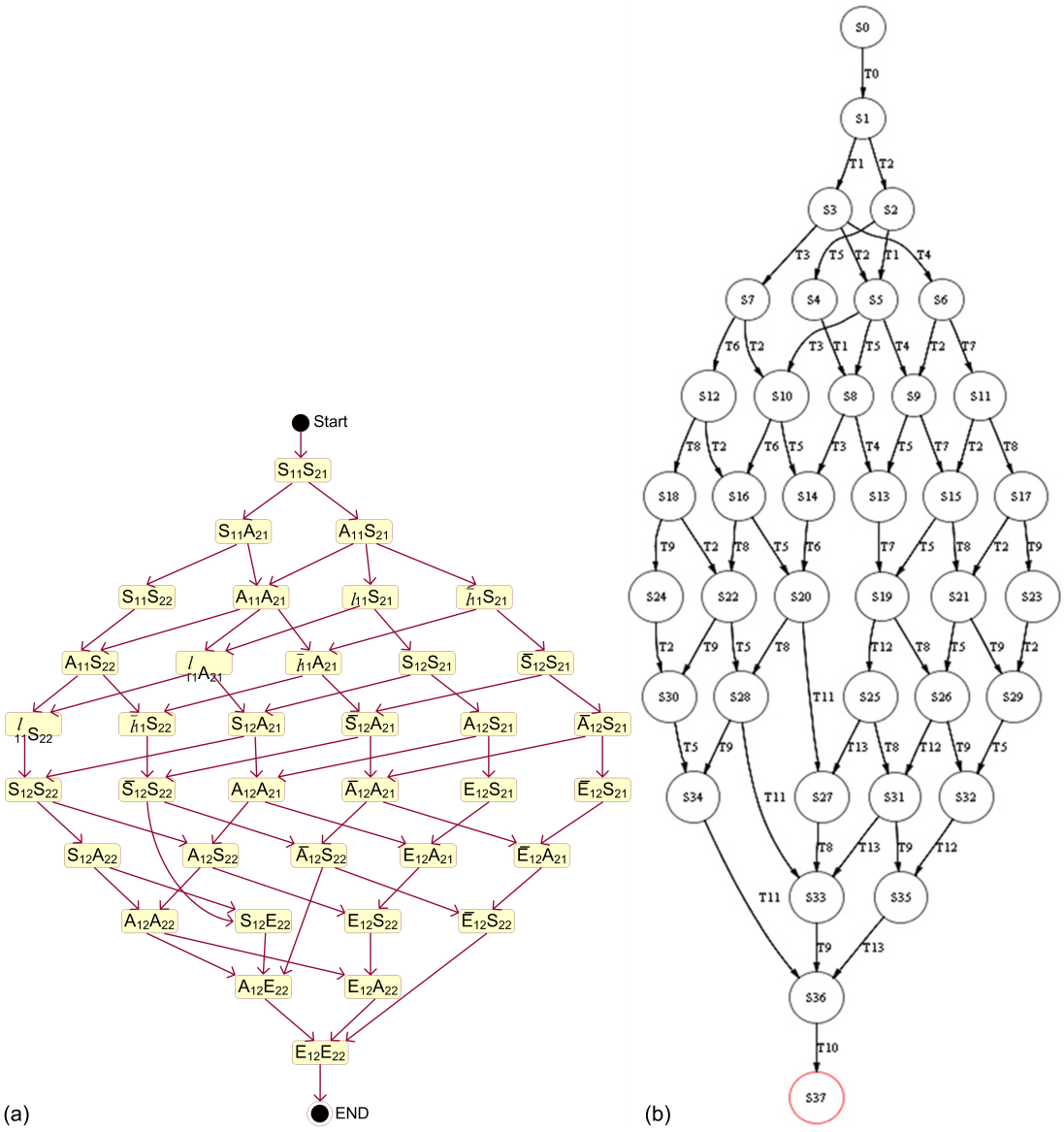


Figure 4.31 State transition systems for Figure 4.30. (a) state machine; (b) reachability graph.

4.6.2 Correctness of the composition process

As the foundation to the formalism of transformation from WS-BPEL to Petri net, the proof of correctness is essential. It contains two steps: correct transformation of basic constructs and correct composition procedure.

Let n be the number of constructs contained in one WS-BPEL process. P_i be the business process with i constructs.

1. Correctness of atomic transformation

   Recall that PN_i is defined in section 4.5 to be the Petri net generated from construct T_i. When $n = 1$, we have only one Petri net PN_1, and $PN_{P_1} = PN_1$. The correctness of PN_1 is gained from the correctness of basic transformation as proved in section 4.4.

2. Correctness of composition procedure

   When $n = 2, PN_{P_2} = PN_1 \circ PN_2$, the correctness of the composition is demonstrated in section 4.6.1.2.

   Assume the composition is correct (the composite Petri net simulates the WS-BPEL process) when $n <= k, i.e., PN_{P_k} = PN_1 \circ PN_2 \circ \ldots \circ PN_k$. For a WS-BPEL process P_{k+1} with $(k+1)$ constructs, $PN_{P_{k+1}} = PN_1 \circ PN_2 \circ \ldots \circ PN_k \circ PN_{k+1} = PN_k \circ PN_{k+1}$.

   Hence $PN_{P_{k+1}}$ is transformed into a WS-BPEL process with only two constructs, PN_{P_k} and PN_{k+1}. Both PN_{P_k} and PN_{k+1} are correct since their size is $\leq$ k. According to the assumption that the composition is correct for WS-BPEL process with $\leq k$ constructs, $PN_{P_{k+1}}$ is correct.

   Thus the correctness is preserved during the inductive composition procedure.

4.7 Evaluation of the Transformation Approach

In chapter 3, we briefly evaluated two existing tools that transform WS-BPEL to Petri net: BPEL2oWFN (94) and BPEL2PNML (104). As pointed out by (17), these two approaches generate abundant places and transitions that increases the complexity of analytic work. The

Table 4.1 Comparison of the three transformations from WS-BPEL to Petri net

Problem	**BPEL2PNML**		**BPEL2oWFN**		**WS-Pro**	
	Places	Transitions	Places	Transitions	Places	Transitions
Readers-Writers problem (11 basic & 5 structured activities)	90	119	119	121	19	23
Dining-Philosophers problem (2 philosophers) (8 basic & 6 structured activities)	92	121	119	121	20	23

examples used by Chitrakar to evaluate the two approaches include a Readers-Writers problem and Dining-Philosophers (2 philosophers) problem. Chitrakar provides detailed BPEL description in (17). The abstract control flow of these two examples is displayed in Figure 4.32. Here we use the same two examples to evaluate our transformation approach. The results are summarized in Table 4.1.

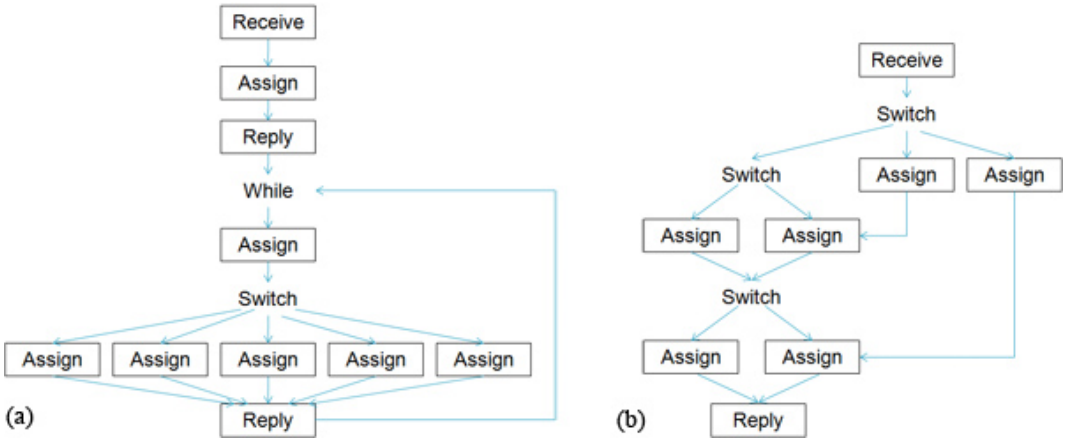


Figure 4.32 Control flow of (a) Readers-writers and (b) Dining Philosophers business processes.

Both BPEL2oWFN and BPEL2PNML create much larger Petri net than our approach, WS-Pro. As an analytic model, Petri net is known to be saddled from state-exposure problem. Many researchers in this area have been devoted to reduce state space. A smaller size is

beneficial for our analysis and computation. Examining BPEL2oWFN and BPEL2PNML, we found that both of them tend to create at least 4 places for basic activities. Moreover, two adjacent Petri nets do not overlap even though the end place of the first one can be regarded as the start place of the next one. In these two approaches, authors add "skip-path" for each activity no matter whether that activity has control flow or now. "skip-path" means this activity will be skipped if:

(1) join condition of control links is false, or

(2) partnerLink do not match, or

(3) corelation set do not match.

However, (1) is unnecessary if the activity does not have control links at all. The irrelevance of (2) and (3) has been discussed in Assumption 2.

Table 4.1 demonstrates the great advantages of our transformation in terms of Petri net sizes. Our transformation generates 3 places for each basic activity and sequential activities share the adjacent places. We only create extra places and transitions when control links are present. Our experiment shows that unnecessary "skip-paths" greatly increase the numbers of places and transitions in the Petri net.

CHAPTER 5. SERVICE COMPOSITION AND ADAPTATION

With the Petri net generated, software designers are able to conduct system property analysis and verification. Rich work has been done in this direction (62)(57)(75)(68) (6). We also reported in (107) our work of using Stochastic Petri net to predict system performance and identify performance bottlenecks.

In this chapter we focus on the other direction: using Petri net to compute the execution plan out from the business process.

Business processes usually include alternative execution paths. This redundancy provides different business options to customers and helps enhance service reliability. It is believed to improve user experience. For example, a travel agency service provides at least three alternative ways to book a vacation to a theme park: (1) call an Airlines service to book air tickets, call a hotel service to book hotel rooms, call a car rental service to reserve a car, and call a theme park service to book park tickets; (2) call another travel agency service such as Expedia to book the whole trip; (3) call another travel agency service such as AAA to book the air tickets, hotel rooms, and a rental car, and then call a theme park service to book park tickets. These alternative execution paths can be specified using If or $Switch$ activities in the BPEL process. If is a better choice in this example because $Switch$ is based on incoming events.

As discussed in section 4.4.2 and 4.4.5, the alternative paths form race conditions in a business process. Process engines may employ different mechanisms to handle these race conditions. The handling of race condition is expressed as "computing the execution plan" in this chapter. The goal of this chapter is to compute such a path based on the Petri net generated using the approach described in Chapter 4.

We build our prototype on the top of PIPE2 (87). PIPE2 is an open source, Java-based

Petri net tool which adopts PNML (Petri net Mark-up Language) standard. The PIPE2 provides a graphic interface for users to create Petri nets, generates reachability graphs, and conduct stochastic analysis.

In our Petri net, weights are used to present performance values (i.e. invocation time and response time). Weights are attached to transitions that are linked to ending places of component services. The path in the Petri net with the least total weight has the best expected performance (shortest response time).

5.1 Performance Metrics

The performance study usually involves a rich set of metrics such as utilization, response time, residence time, queue length, and throughput, etc. In this thesis, we only consider the metrics related to time observation that is regarded as the most important factor to illustrate user satisfaction on online systems (12). We use two metrics: "invocation time" and "response time". They are defined as follows.

Definition 4. *Invocation time is the length of time required for invoking a service.*

Definition 5. *Response time is the length of time that a user must wait from the instant that they invoke a service to the instant that they receive the response to that request.*

"Invocation time" usually includes the time spent on packing/unpacking SOAP messages, processing/validating XML documents, data retrieval, etc. "Response time" depends on many factors such as operation of services, database visits, network delays, resources utilization, etc.

5.2 Computation of Execution Plan

5.2.1 Algorithms

Algorithm 6. *String COMP-PATH(PN pnet)*

Begin

COMPRESS(pnet)

Remove loops in *pnet* using *LoopRemoval*(*pnet*)

Path ← $COMP\text{-}PATH$(*pnet*)

Return Path

End

This algorithm has five steps.

Step 1 Remove false control links. They do not affect performance analysis and create unnecessary places and transitions. *COMPRESS()* function helps reduce the size of the graph.

Step 2 Augment the Petri net with performance data, invocation time and response time. The performance data is reasoned out from historical data or specified by the service level agreement. We can use data mining techniques when a large amount of historical data is available, otherwise we can conduct re-sampling and non-parametric statistic estimation techniques. When no historical data is available, we use the performance specification from the Service Level Agreement (SLA) (25).

Step 3 Remove all the loops in the Petri net. The *LoopRemoval* algorithm uses Control Flow Graph (CFG) algorithms to identify loops and depends on statistic mean of iterations to perform loop removal.

Step 4 Compute the optimal execution plan using $COMP\text{-}PATH()$.

In this algorithm, only the last step is runtime computation.

5.2.1.1 Loop Removal

Our preliminary experiments show that the processing of loops (the Petri net may have nested loops) caused steep increase in runtime computation if the graph is cyclic. If we can remove all the loops before service execution, we can greatly reduce the runtime computation. To achieve this, we designed a statistical loop removal technique utilizing historical data (Step 3 in the algorithm above).

The algorithm of removing loops has two steps: loop detection and loop removal. The Petri nets studied in this research all have starting node and ending node. They can be structurally treated as Control Flow Graph. Therefore, we can use the existing CFG theories to implement loop detection.

Before removing loops (in fact, removing the back-edges), we need to modify all the weights contained in this loop so that they reflect the correct estimation. To do this, we calculate the mean iteration $\bar{I}$ for each loop and multiple $\bar{I}$ to all the weights contained in that loop.

The details of this algorithm is shown as below.

Algorithm 7. *Graph LoopRemoval(Graph G)*

Begin

/∗**Part I: Identify loops**∗/

/∗All the loops are either mutually disjoint or one is completely contained in the other∗/

Find all loops and store then in the set $LOOP$ using Ramalingam's enhanced algorithms (90)

/∗**Part II: Remove loops**∗/

For each loop $Loop_{yx}$ in $LOOP$ **Do**

Calculate the mean number of iterations $\bar{I}$ based on the rate of (y, x)

Multiply all the weights in this loop by $\bar{I}$

/∗Remove the loop∗/

Remove the back-edge

End

(1) Loop detection

In CFG theory, one of the most widely known algorithms for loop detection is to use "dominators", demonstrated in (1). The basic concepts of this algorithm are described in Algorithm 8. Havlak (45) proposed another algorithm by modifying Tarjan's algorithm proposed in 1974 and 1983 (101). This algorithm can detect loops for reducible and

irreducible graphs. In 1999, this algorithm was modified again by Ramalingam to improve its time complexity (90). With all these efforts, this algorithm can identify loops from a CFG in almost linear time. Another popular algorithm in the CFG research was proposed by Sreedhar and his colleagues (98). It was enhanced by Ramalingam to make it run in almost linear time too (90).

Our *LoopRemoval* utilizes the existing optimized algorithms to identify loops. The time complexity, as reported by the above researchers, is almost linear. We will use $O(m)$ as the running time for loop detection in the followings, where $m = |IN \cup OUT|$.

Algorithm 8. *Aho-Sethi-UllmanAlgorithm(Graph G)*

Begin

/∗**Part I: Find dominators for each place**∗/

/∗In a Control Flow Graph $(V, E, Start, End)$, a node x dominates y, if every path from *start* to y has to pass through x. $n = |V|$ and $m = |E|$. ∗/

$D_{start} = \{Start\}$ //D_i is the set of dominator for node i

For each $i \in V$ and $i \neq start$ **Do**

$D_i = V$

End For

While change **Do**

$change = false$

For each node **Do**

$tempD_i = i + D_{j1} \bigcap D_{j2} \bigcap \ldots \bigcap D_{jm}$

where $j1 \ldots jm$ are the predecessors of i.

If $tempD_i \neq D_i$ **Then**

$D_i = tempD_i$

$change = true$

End If

End For

End While

/∗**Part II: Identify loops**∗/

For each edge (y, x) **Do**

If (y, x) is a back-edge **Then**

/∗A back-edge (y, x) is an edge from y to x where x is y's dominator∗/

$Dby_x = \{$all the nodes that are dominated by $x\}$

/∗Find all the nodes inside the loop∗/

$Loop_{yx} = Dby_x \bigcap \{$nodes that can reach $y\}$

Add $Loop_{yx}$ to $LOOP$ /∗$LOOP$ is the set of loops∗/

End If

End For

End

(2) Loop removal

We have two types of loops: stochastic loop and deterministic loop. *While* and *RepeatUntil* are of the first type, sequential *ForEach* is a deterministic loop. For a deterministic loop we know the number of iterations beforehand, which is $\bar{I}$. For stochastic loops, we need to calculate its mean value. Let the probability of staying in the loop be p, which is annotated as the rate of the transition *loop* in the Petri net. The number of iterations till the first time when *loop*'s competitive transitions are fired follows *Geometric* distribution. By the properties of *Geometric*, we know that its expected value (mean) is $\frac{1}{(1-p)}$. p is set based on the harmonics of statistically analyzed historical data, Service Level Agreement (SLA) specifications, and performance requirements.

Now we analyze the time complexity of the loop removal process. For each loop, both computing the mean iteration $\bar{I}$ and removing the back-edge take constant time. Modifying weights contained in each loop takes $O(m)$ time in worst case, where $m = |IN \cup OUT|$. Putting these operations together, the required time to process one loop in worst case is $O(m)$. The number of loops is no more than $O(|T|)$, where $|T|$ is the number of transitions in the original Petri net. Therefore, the process of removing loops takes $O(m * |T|)$ running time in worst case.

(3) Time complexity

From the analysis above, the loop detection process runs in almost linear time and the loop removal process runs in $O(m*|T|)$ time in worst case. As a result, Step 3, "Removing loops from the Petri net" takes $O(m * |T|)$ running time.

5.2.1.2 Path Generation

Here we add a new variable called *distance* that represents the least total weights collected along the path from the *start* place to a place or a transition.

The *distance* variable can be associated with both places and transitions. In other words, we are searching for a shortest path (in terms of time) from the *start* place to each other place.

The initial value of *distance* is 0 for all the places and transitions. After the computation, the *distance* on the *end* place is the total expected response time of the optimal execution path.

Algorithm 9. *String COMP-PATH(PN pnet)*

Begin

For each place p **Do**

Set $distance_p = 0$

End For

For each transition t **Do**

Set $distance_t = 0$

End For

While change **Do**

$change = false$

If find a place, p that satisfies the following

(1) p's distance = 0, and

(2) p has predecessor(s), and for each of p's predecessor place pp,

$distance_{pp} = 0$ if $pp = start$; otherwise $distance_{pp} > 0$,

Then

change = true

/*Updating the distance for this place*/

For each transition i that has p as an output place **Do**

$distance_i = max\{distances\ of\ input\ places\} + weight_i$

End For

Set $distance_p = min\{distance_i\}$

Set the $backtrace = the\ chosen\ transition$

End If

End While

End

The algorithm 9 scans the graph and updates all the places until the end place is updated. We only update a place when all of its predecessors have already been updated. When there are multiple input arcs, the updating obeys the following rules. Figure 5.1 illustrates the rules.

- For places:

  If there are multiple input transitions (If and $Pick$ situations), then choose the transition with the minimum distance.

- For transitions:

  If there are multiple input places ($Flow$ situation), then choose the place with the maximum distance.

We also set a variable *backtrace* to trace the chosen predecessor. This way, after computation, we can output the whole optimal execution path.

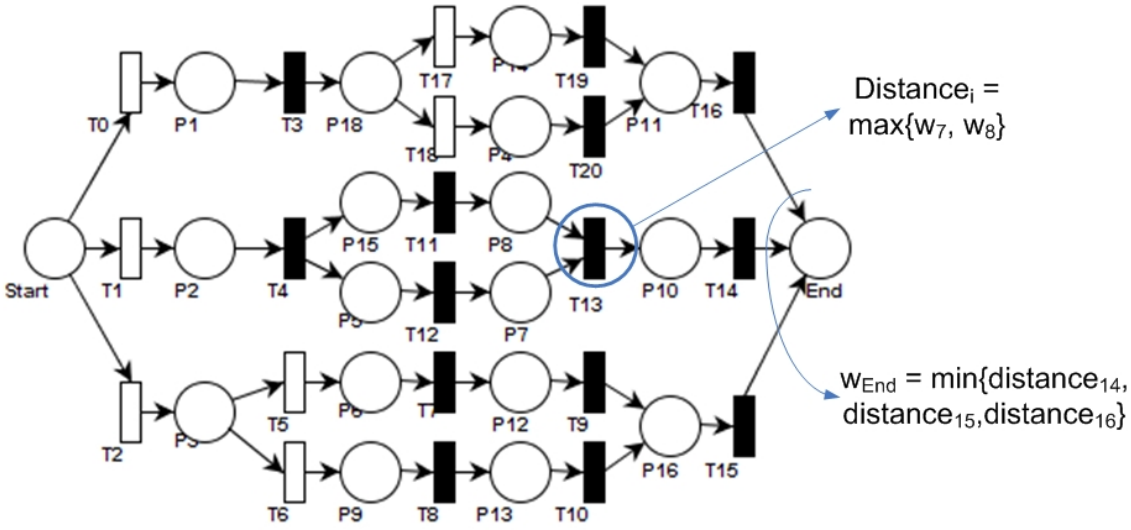


Figure 5.1 Updating rules

(1) **Correctness**

In Algorithm 9 the requirements (1) and (2) decide that each place is updated only once. We can prove that this is a reasonable restriction.

Claim 1. *According to Algorithm 9, each place needs to be updated only once.*

Proof. Let n = the number of places in Petri net *pnet*.

When $n = 1$, the *start* place is updated once.

Assume Claim 1 is correct when $n < k$. In other words, when the Petri net has $< k$ places, each place only needs to be updated once.

When $n = k$, assume place p_i needs to be updated twice. That means at least one of p_i's predecessors, p_a, was updated again after p_i's first update. Considering the Petri net without p_i, it has size of $k - 1$. However, one of its places, p_a was updated more than once. This conflict suggests that Claim 1 applies for Petri net with size $\geq k$. □

(2) **Time Complexity**

In Algorithm 9, we need to update all the places ($O(|P|)$), and each place may have $O(|T|)$ transitions. Therefore, Algorithm 9 has runtime computation of $O(|T| * |P|)$.

5.2.2 To complete the FireHazard example

This chapter describes the process of computing an optimal execution plan when there are alternative paths (more specifically, there are If or $Pick$ constructs in the business process description). We have two examples demonstrated in chapter 4, $SeeMovie$ and $FireHazard$. The $SeeMovie$ example does not include any If or $Pick$ activity. There is no optimization issue in that example. Hence, we work on the $FireHazard$ example to demonstrate how Algorithm 6 works. The $Firehazard$ example is described in section 4.5.2.2. Its WS-BPEL description and generated Petri net $PN_{firehazard}$ are shown in Appendix C and Figure 4.18, respectively.

$PN_{firehazard}$ does not have a control link, hence we skip the $COMPRESS$ module. The Petri net generated in Figure 4.18 is augmented using the expected time information. The Petri net with performance information is shown in Figure 5.2. The response time of the first component activity *Receive* is $300ms$. Performance data of other activities are denoted on the Petri net similarly. Note the "invocation time" is associated to the $Invoke$ activities only. In Figure 5.2, transitions $jt21, jt22, jt23, jt31, jt32, jt33$ (corresponding to $Invoke$ activities: *Alarm*, 911 *call*, *Sprinkler*, *Smoke detector*, *Chemical sensor*, *Temperature sensor*) are associated with weights that represent "invocation time".

There is one loop in $PN_{firehazard}$, going from place $e4$ to place $e1$ through the transition *loop*2. The firing rates of *loop*2 and $te4$ are 0.7 and 0.3, respectively. In Petri net, exponentially distributed firing rates specify the firing delay. When both t_1 and t_2 are enabled with firing rates r_1 and r_2, the probability of firing t_1 is $\frac{r_1}{r_1+r_2}$. Therefore, the probability of firing *loop*2, i.e., the probability of staying in the loop, is 0.7.

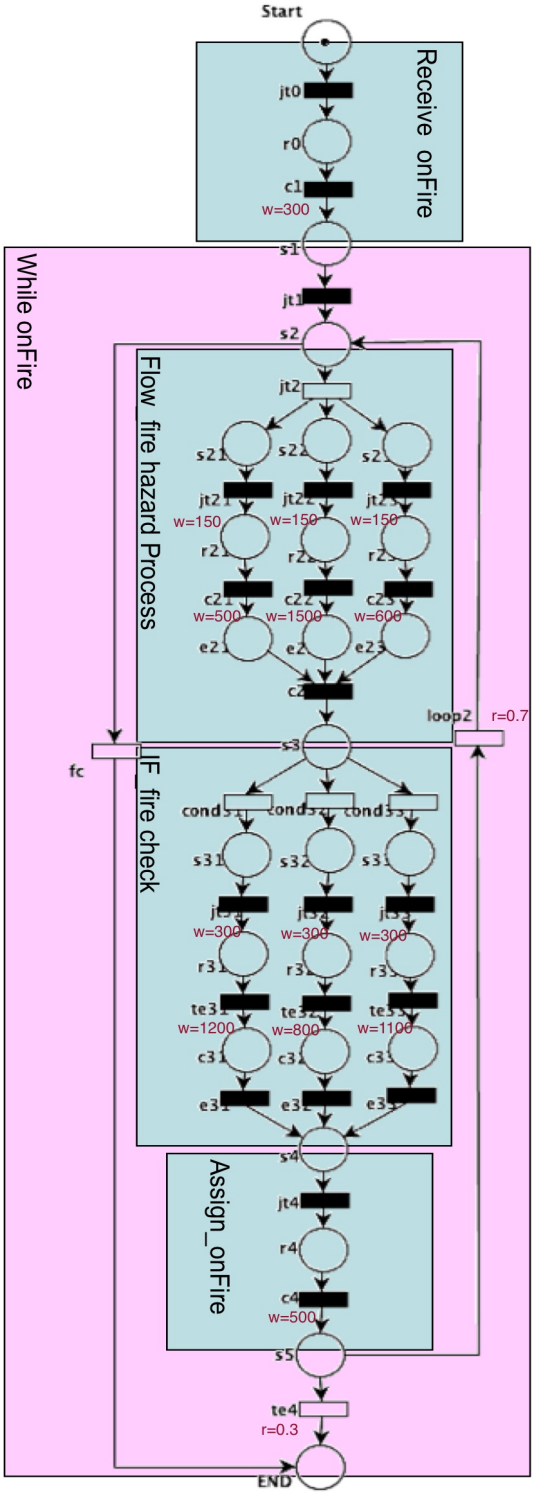


Figure 5.2 $RG_{firehazard}$ augmented with performance data.

The *LoopRemoval* algorithm detects this back-edge of $loop2$ and identifies all the nodes included in this loop. Then the algorithm calculates the mean number of iteration $\bar{I} = \frac{1}{(1-r_{loop2})} \approx 3.33$ (the mean of Geometric Distribution). All the weights associated with transitions in the loop are timed by $\bar{I}$. Finally, the *LoopRemoval* algorithm removes the back-edge $loop2$.

Based on the acyclic Petri net, Algorithm 9 computes an execution plan with optimal response time. The optimal execution plan is represented as a sequence of transition firing: $jt1, c1, jt2, jt21, c21, jt23, c23, jt22, c22, c2, cond32, jt32, te32, c32, jt4, c4, te4$, which represents the activities in the sequence of *receive(onFire), While{Flow, Invoke(Chemical sensor), Assign(onFire)}*.

5.3 Service Adaptation

Service adaptation mechanism applies when performance failure is captured. The performance failure is the service failure of not meeting the performance requirements. In this thesis, it specifically means failure to meet the response time requirements. Performance failure happens frequently in the heterogenous Web Services framework. The possible causes includes network failure, service relocation, request overload and operation rename, etc. It is of particular interest to researchers to develop a mechanism that can re-plan the service composition so that the service can be delivered without interruption. We present our adaptation algorithm in Figure 5.3.

In the service composition environment, service executor is a software component to execute the composite service (generating service bindings, packing SOAP messages, monitoring service execution, etc.). When the service executor finds that a certain component service has a performance problem, it calls for the adaptation algorithm to act.

5.3.1 Algorithm

This algorithm can be applied on the acyclic reachability graph or directly on the acyclic Petri net. There are two considerations in the design of adaptation algorithm.

(1) We set backup candidate services for each activity. This is particular meaningful to

```
Name: adaptPath
Input: int problematicTask, int pathID, currentExecutionTime
Output: executionPath newPath

Read in compositionGraph
newPath ← the path with ID equal to pathID
newPath.problematicTask.meanExecutionTime = alterCandidateMeanTime
newtotalExecutionTime = currentExecutionTime + restEstimationTime
If newtotalExecutionTime < PathTimeout then
   return newPath
else
   currentNode = problematicTask
   Do
         branchNode ← first node with alternative path backward from currentNode
         start of compositionGraph ← branchNode
         localNewPath ← optPath(compositionGraph)
         localNewPath.totalTime += currentExecutionTime + defaultAdaptationCost
   Untill ((currentNode==start) OR (localNewPath.totalTime <= PathTimeout))
   If (localNewPath.totalTime <= PathTimeout)
      return failure
   else
      return localNewPath
```

Figure 5.3 Adaptation algorithm

business alliances. When performance problem arises, we check alliances for backups. If replacing component service provider does not solve the problem, we re-compute the execution path.

(2) We do performance optimization under the consideration of business cost. Since each service invocation has financial cost, we do re-planning in a hill-climbing way. That means we try our best to keep the services that have already been executed. In this case, the adaptation plan may not be the best solution but is a near optimal solution with the least budget lost.

5.3.2 To complete the FireHazard example

In section 4.5.2.2, we compute an optimal execution path of "*receive(onFire)*, *While*{*Flow, Invoke(chemical sensor), Assign(onFire)*}". Suppose that a performance exception is captured with the *Invoke(chemical sensor)* service during the execution of this composite service $FireHazard$ (e.g., the expected response time of 1.5 sec has been exceeded). Then, the service executor calls for the adaptation algorithm shown in Figure 5.3. The adaptation algorithm replaces the service provider for the *Invoke(chemical sensor)* activity and returns the updated

composition plan to the service executor.

Unfortunately, all the service providers for *Invoke(chemical sensor)* are not responding due to the damage caused by the fire. Then the adaptation algorithm was called again for re-planning. The adaptation algorithm then checks the composition graph for alternative execution paths (to avoid the *Invoke(checmical sensor)* activity). *Invoke(temperature sensor)* is chosen as the path to replace the *Invoke(checmical sensor)* activity.

5.3.3 Simulation

The $FireHazard$ example has only one If activity that contains three basic activities. This example is too small to evaluate the effectiveness of the adaptation algorithm. We use simulation to evaluate the effectiveness of the adaptation algorithm. Figure 5.4 depicts an acyclic graph we used in the simulation. Each component service in the graph shown in Figure 5.4 has two values. The first value is its average response time. The second value is the average response time of one of its backup services (As stated in previous section, each activity has backup candidate services).

In Figure 5.4, the dotted path is our original execution path. Suppose now node *t13* has performance exception. Replacing the service provider can not solve the problem. Then we move backward to the first node with an alternative path, which is *t6*. All the nodes between *t6* and *t13* have been executed, and we now discard their execution. We set *t6* to be starting point and try to find another path from *t6* to the end. If we cannot find such a path to satisfy performance requirements, we repeat the backward motion over and over till we reach the starting point $t1$.

As described above, the adaptation algorithm handles service exceptions. In a Web Services environment, service consumers have no way to differentiate the causes of service exceptions. They can not tell whether the service stops functioning or it is still functioning but with performance delay. The general strategy is to consider both cases as "service failure". Here we define service failure as below.

Definition 6. *Component deadline: a component deadline is the predefined response time for*

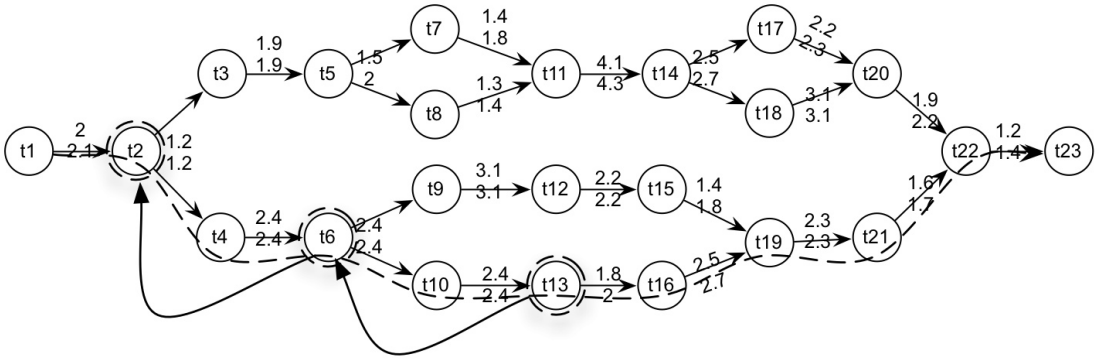


Figure 5.4 Service adaptation: acyclic execution graph and adaptation process

a specific component service.

Definition 7. *Global deadline: a global deadline is the predefined response time for a composite service.*

Definition 8. *Component service failure: a component service is considered to be failed if the service execution passes a predefined deadline.*

Definition 9. *Composite service failure: a composite service fails if its execution passes its predefined global deadline or the composite service fails to deliver its functionalities.*

In order to analyze the effect of adaptation algorithm, we set the service deadlines according to probability coverages. For example, if a deadline is set according to a 95% coverage, then we expect the service to finish before the deadline with the probability of 95%. **In other words, the probability coverage sets the component failure rate.** A 90% probability coverage sets the component failure rate to 0.1.

Component deadlines work with adaptation. As described above, when the execution of a component service exceeds its deadline, the adaptation algorithm kicks in and re-plans the service composition. In other words, the component deadlines decide when the adaptation algorithm should apply.

Since we set component deadlines according to probability coverages, the component deadlines might be large values (especially for Exponential distribution). In reality we usually have

a global deadline for the composite service set according to the performance requirements. *The global deadline is a much lower bound than the sum of all component deadlines in most cases.* Therefore, even though with probability coverage 1, in which case we should have failure rate 0, we might still have service failure due to the bound of the global deadline.

The machine used for simulation has the following configurations.

- CPU/Memory: Intel(R) Pentium(R) 4 CPU 3.20GHz, 3GB RAM
- OS: Red Hat Enterprise Linux 4, Linux 2.6.9
- Virtual Machine: Sun Java JDK 5.0, version 1.5.0_08

The simulation is conducted with the following specifications.

- Iterations conducted to generate each average value: 10,000,000
- Probability distribution used to generate response times: Exponential, truncated Normal, Gamma, and Erlang (which is a special Gamma distribution). The Normal distribution is truncated to eliminate negative response time. In the rest of the chapter, the term "Normal" is used instead of "truncated Normal" to ease presentation.

The purpose of the simulation is to evaluate the effectiveness of the adaptation algorithm. The simulation results are described in two parts: average response time and average failure rate.

5.3.3.1 Average total response time

The first job of this simulation is to evaluate the performance of our Adaptation mechanism. The Adaptation mechanism introduces overhead to the service execution time. It includes the time for adaptation computation itself, and the time spent on executing alternative services and alternative paths. In oder to evaluate the performance of our Adaptation mechanism, one major task is to evaluate whether the overhead added by the Adaptation mechanism is tolerable in the composite service. We want to compare the total response times of the composite service when with adaptation and without adaptation.

In this simulation, all the component services respond independent from each other. The composite response time is the sum of all the component response times. According to the statistical theory, the mean response time of the composite service is the sum of component means. Therefore, the mean response time of the composite service composed from Figure 5.4 will be in the range of $[19.8, 20.8]$ depending on which execution path is chosen.

The average total response time of the simulation is shown in Figure 5.5. The comparisons are shown based on different probability distributions for the component services. Among the three distributions, Exponential has the longest tail and Normal has limited scale.

When the adaptation algorithm is not used, we allow all the component services to finish their execution no matter how long that execution takes. In other words, there is no deadline for component services. Therefore, the probability coverage does not affect the total response time (the probability coverage is used to set component deadlines). This is demonstrated in Figure 5.5. The total response time of the composite service is almost constant when adaptation algorithm is not used.

When the adaptation algorithm is used, each component service has a deadline. Once component services pass their deadlines, the adaptation algorithm is triggered. Revisiting Definition 9, we know that the composite service may abandon its execution when its component services fail and no valid alternative execution plan can be found. However, in this specific simulation, we allow the failed composite service to finish. This way, we can analyze the real total cost of the composite service, thus evaluate the performance of the Adaptation mechanism.

In Figure 5.5, a curve is presented when the adaptation algorithm is used. This curve illustrates that the adaptation algorithm adds overhead to the total response time. This overhead contains the cost of adaptation algorithm and the execution of alternative services/paths. With the increasing of probability coverage, the number of failed component services drops, which results in fewer invocations of adaptation algorithm and less overhead.

The overhead added by the adaptation algorithm is small. With the adaptation algorithm, the total response time is increased by 2.85% at probability coverage 90%.

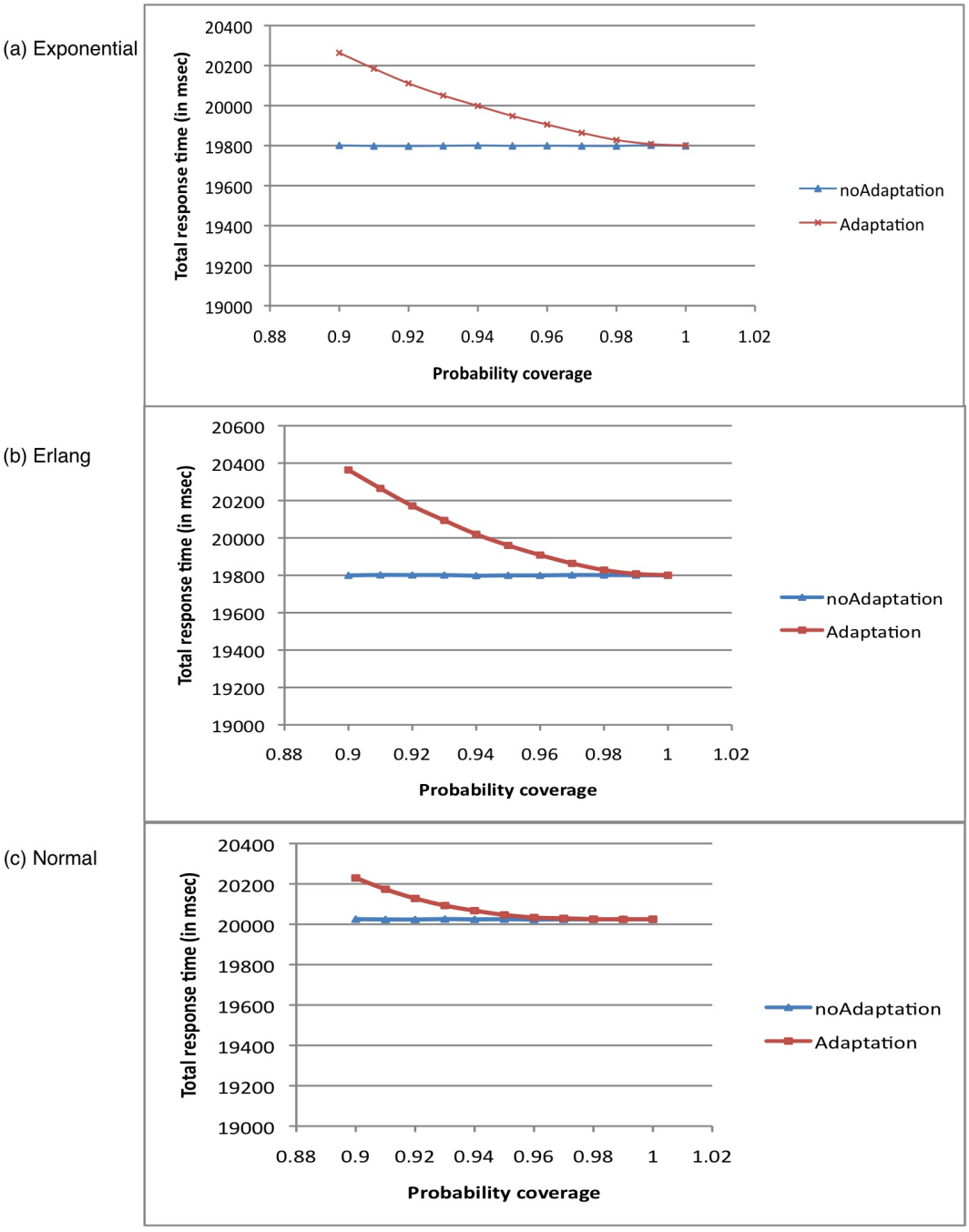


Figure 5.5 Average response time: Adaptation vs. noAdaptation

To check the impact of the adaptation algorithm on the total response time, we demonstrate the histogram of the total response time at probability coverage 90% and 99.5%, illustrated in Figure 5.6 and Figure 5.7, respectively. These two graphs show that the histograms of the two cases, "NoAdaptation" and "Adaptation", are similar. They have the same scale. The "Adaptation" curves are more right-skewed with thicker tails. They also have less mass in the "center". This phenomenon is better demonstrated in Figure 5.6. Figure 5.7 has very low component failure rate. Adaptation mechanism is less invoked. Therefore the two histograms of "NoAdaptation" and "Adaptation" almost match.

5.3.3.2 Failure rate

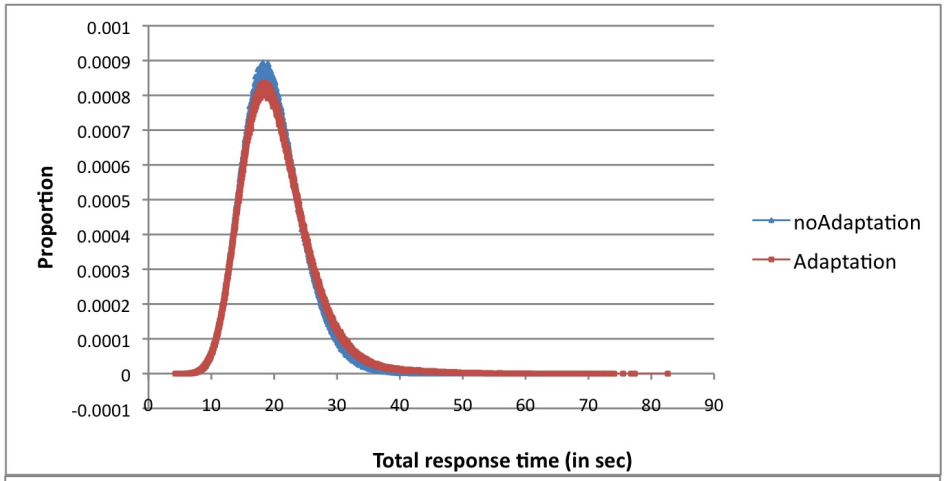
In this section, we evaluate the failure rate of the component service when with adaptation and without adaptation. As discussed above, a service is considered to be failed once its execution exceeds the predefined deadline. Besides, any component service failure causes the composite service to fail when the adaptation mechanism is not in use. Below we report our simulation results in two cases: with global deadline and without global deadline.

1. Without global deadline

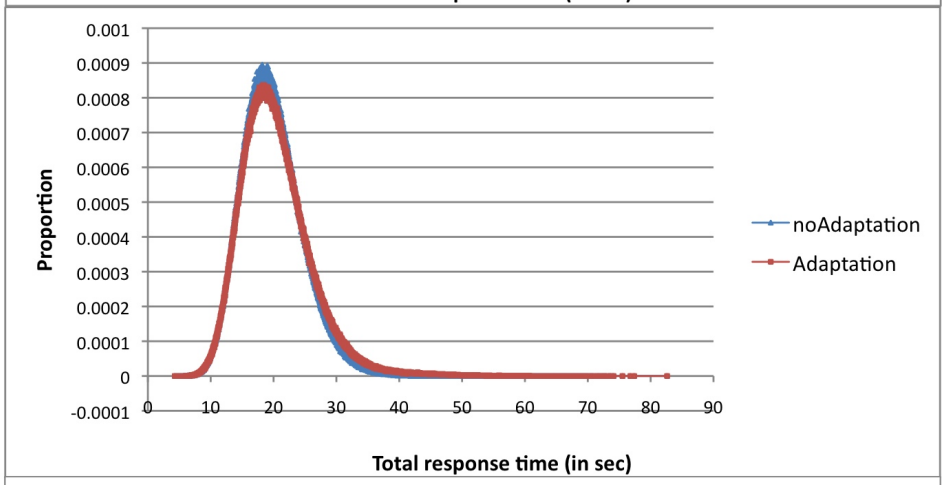
In this simulation, we set the global deadline to a very large number $Long.MAX\_VALUE = 2^{63} - 1$.

Probability distribution is not an impact factor in this simulation. It only affects component deadlines. For example, with a $Exponential(20.0)$ distribution, when the probability coverage is equal to 0.995, the component deadline is 105.966. If with a $Erlang(10, 2.0)$ distribution, this component deadline should be 39.997. However, since we do not have a global time boundary, we do not need to worry about the values of component deadlines. All we care is whether a component service fails or not. The probability of this failure is pre-defined (the probability coverage) for any probability distribution. Therefore, both *Exponential* and *Erlang* generates the same results in this simulation. Below we only display the simulation results based on Exponential distribution.

(a) Exponential



(b) Erlang



(c) Normal

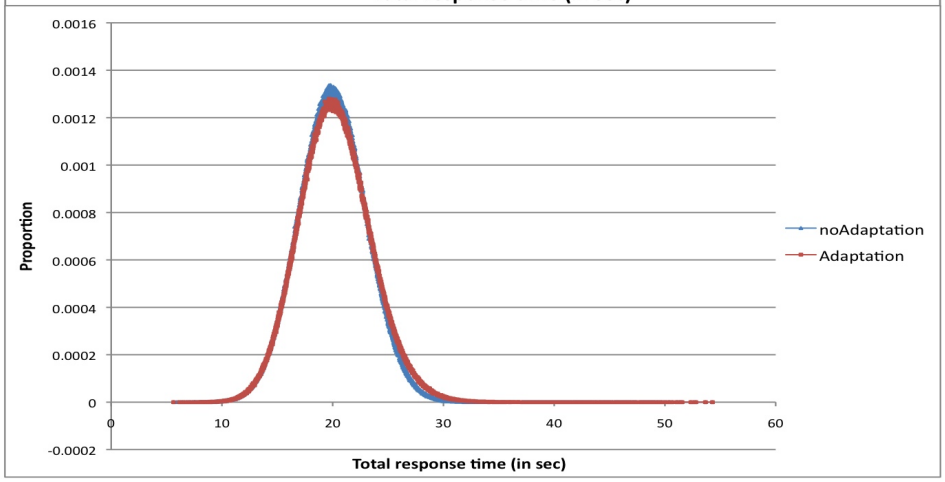


Figure 5.6 Histogram of the total response time at 90% probability coverage

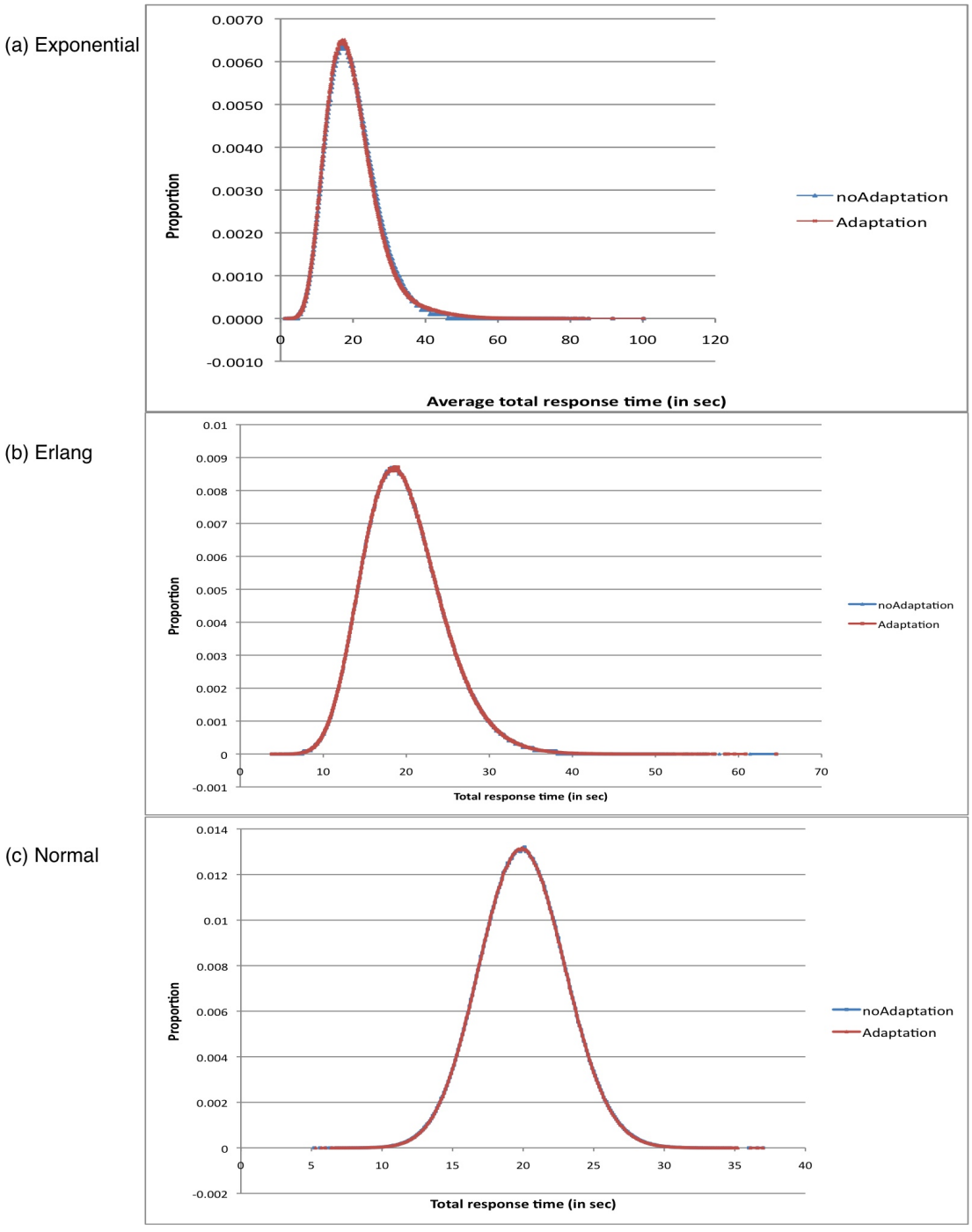


Figure 5.7 Histogram of the total response time at 99.5% probability coverage

We set the probability coverage from 0.9 to 0.995 and simulate the execution of the composite service. The failure rates of the composite service according to different probability coverage are depicted in Figure 5.8.

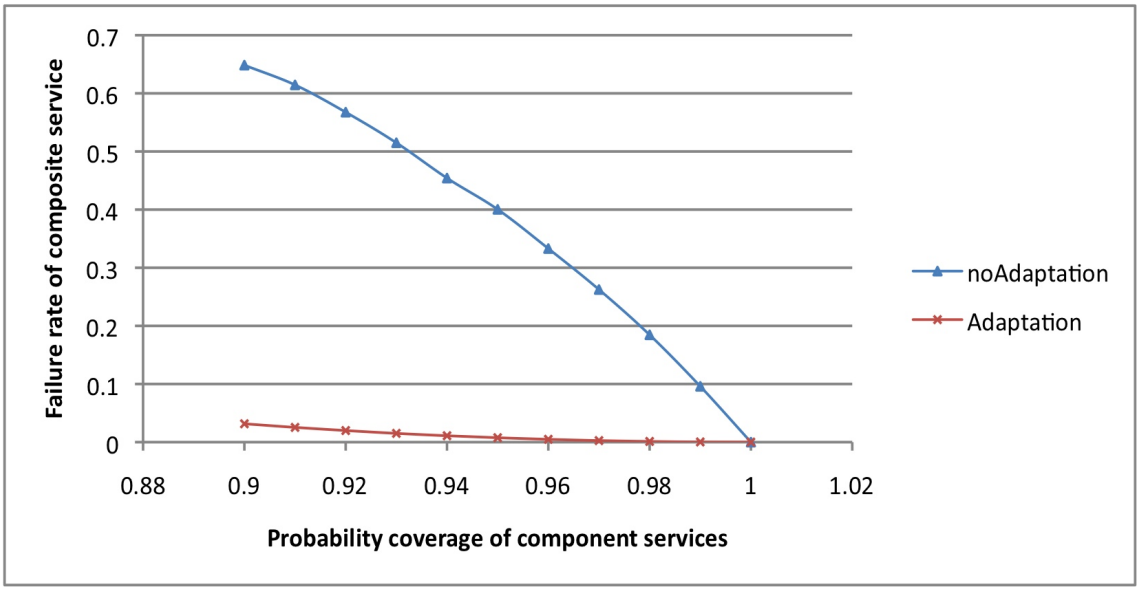


Figure 5.8 Average failure rate without global deadline: Adaptation vs. noAdaptation

This figure shows that the failure rate drops for both "noAdapation" and "Adapation" cases when the probability coverage increases. This phenomena conforms to the theory: higher probability coverage results in long deadlines, which lead to fewer failures.

At coverage 90%, the failure rate of "noAdapation" is 0.6559, as displayed in Figure 5.8. This can be easily explained by probability theory. For 10 sequential jobs, if each job has 90% chance to succeed, then the probability of all successful execution is $0.9^{10} = 0.3486$. Hence, the probability of composite service failure is $1 - 0.3486 = 0.6513$, which is close to our simulation result, 0.6532.

Contrast to the high failure rate of "noAdapation", the failure rate for the "Adapation" case is very low, ranging from 0 to 0.04. The failures for "Adaptation" appear only when there are enough failures in the graph shown in Figure 5.4 such that the adaptation algorithm can not find alternative paths. The high failure rate for "noAdaptation" represents the "single point

failure" problem in business composition using WS-BPEL. The composite service is aggregated by multiple component services. Any fault of the component services (content fault or timing fault) causes composite service fault. In reality, the high failure rate forces the execution engine to frequently abandon the execution and re-invoke the service. This is unacceptable in the business world. Our simulation results demonstrate that a remedy mechanism is necessary in the Web Services environment.

2. With global deadline

The global deadline is usually set by the performance requirements (or expected performance). The sum of component deadlines may be greater than the global deadline. Hence, different probability distribution may affect our simulation results as the probability distribution affects component deadlines. In this simulation, we used three probability distributions: Exponential, Erlang, and Normal. The shape parameter k of Erlang is set to 2 so that this distribution is more right-skewed (positive-skewed with a longer upper tail).

(1) **Firstly, we check how the global dealdine affects the "noAdaptation" case.** The probability coverage sets the failure rates for all the component services, regardless which distribution is used. We compare the failure rates of "noAdaptation" in two cases: no global deadline and 20 seconds global deadline. The results are shown in Figure 5.9.

When there is no global deadline, the failure rate is the same for all the distributions. As explained above, this is identical component failure rates.

We then set the global deadline to 20 seconds, which is roughly the expected mean response time (by adding all the mean response time of component services together). As demonstrated in Figure 5.9, when the probability coverage is low (short deadlines), the global deadline has slight effect on the failure rate of the composite service (only increased the failure rate by 0.03% at probability coverage 90%). With higher probability coverage, each component service has a longer deadline. This allows earlier services plenty of time to finish while causing latter services to fail due to the global deadline. We call this effect as "Loose head, tight tail". This effect is apparent when we have a short global deadline: in the Figure 5.9, when probability coverage approaches 1.0, the curve with 20 seconds

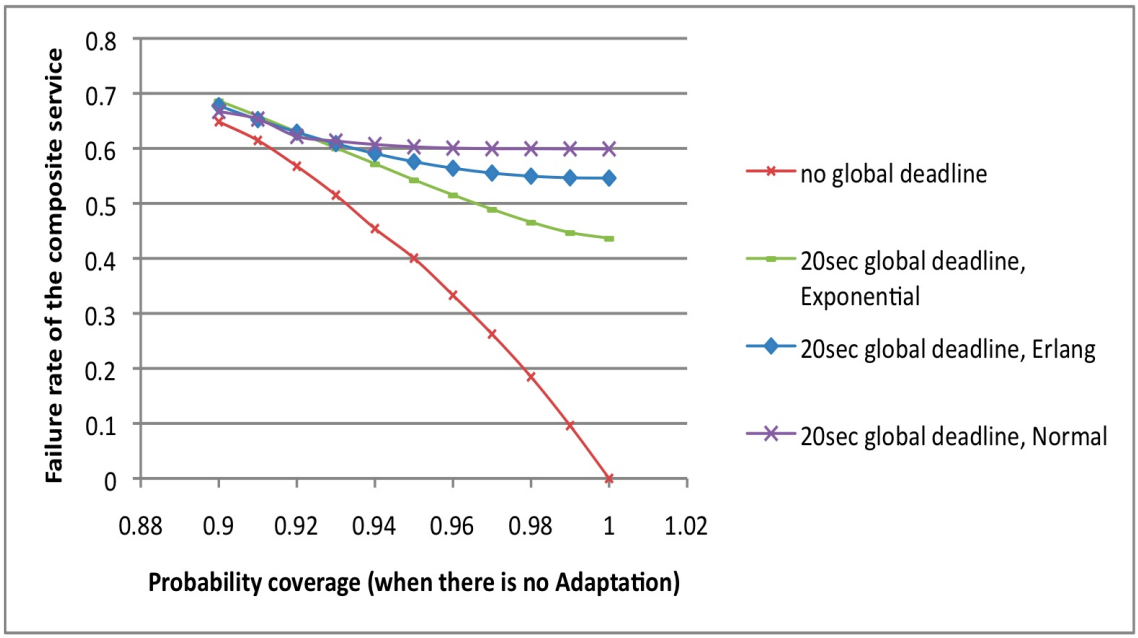


Figure 5.9 Failure rates for "noAdaptation". No global deadline, 40sec global deadline and 20sec global deadline

global deadline has higher failure rate than the one without global deadline.

Besides, we observed higher failure rates for distributions with more left skewness. Comparing the CDF functions of the three different distributions, we see that Normal generates the lowest cumulative distribution value when $x \leq 20.27$, where x is the total response time. That indicates more service failures for Normal distribution.

(2) **Secondly, we check how the global deadline affects the "Adaptation" case.** Figure 5.10 displays the comparison between different deadlines according to different probability distributions.

It is intuitive that shorter global deadlines result in higher failure rates. Long global deadlines accommondates the adaptation overhead better than short global deadlines. This phenomena is confirmed in Figure 5.10, where a 20 seconds global deadline causes a sharp increase in the failure rates of "Adaptation" case.

(3) **Thirdly, we check how the probability distribution affects the "Adaptation" case.** We have already discussed that when the global deadline is not present, the

probability distribution does not affect the failure rate of the composite service. Below we show the impact of probability distribution on the failure rate when global deadline is set.

From Figure 5.10 we can see that the change of probability coverage does not impact Erlang and Normal as much as it impacts Exponential. This is because Exponential has much higher component deadlines than the other two distributions for the same probability coverage. The change of probability coverage also results in bigger change in Exponential's component deadlines. This phenomenon can be verified from the inverse cumulative distribution functions. The component deadlines determine when Adaptation algorithm kicks in. As discussed at the beginning of this subsection, long component deadlines result in the "Loose head, tight tail" problem. A "Loose head, tight tail" case increases the failure rate. This is why the failure rate drops for Exponential when the probability coverage increases. The decreasing in the failure rate illustrates the effectiveness of the Adaptation mechanism.

Instead, in Erlang and Normal cases, especially Normal that has narrower scale, the change of probability coverage does not impact the local deadlines as much as for Exponential. Accordingly, the component deadlines do not change that much with the increasing of probability coverage. Therefore, the failure rates for Erlang and Normal, especially Normal, do not have obvious decrease.

From the discussion above, we can see that the adaptation algorithm greatly helps reduce the failure rate of the composite service, while adding tolerable overhead to the total response time. Its effectiveness in reducing the failure rate drops with the decreasing of the global deadline. Different distribution of the component service response time may affect the failure rate of the composite service. Therefore, when selecting the service provider for a certain task, it is benefitial to analyze the operational profile of candidate providers.

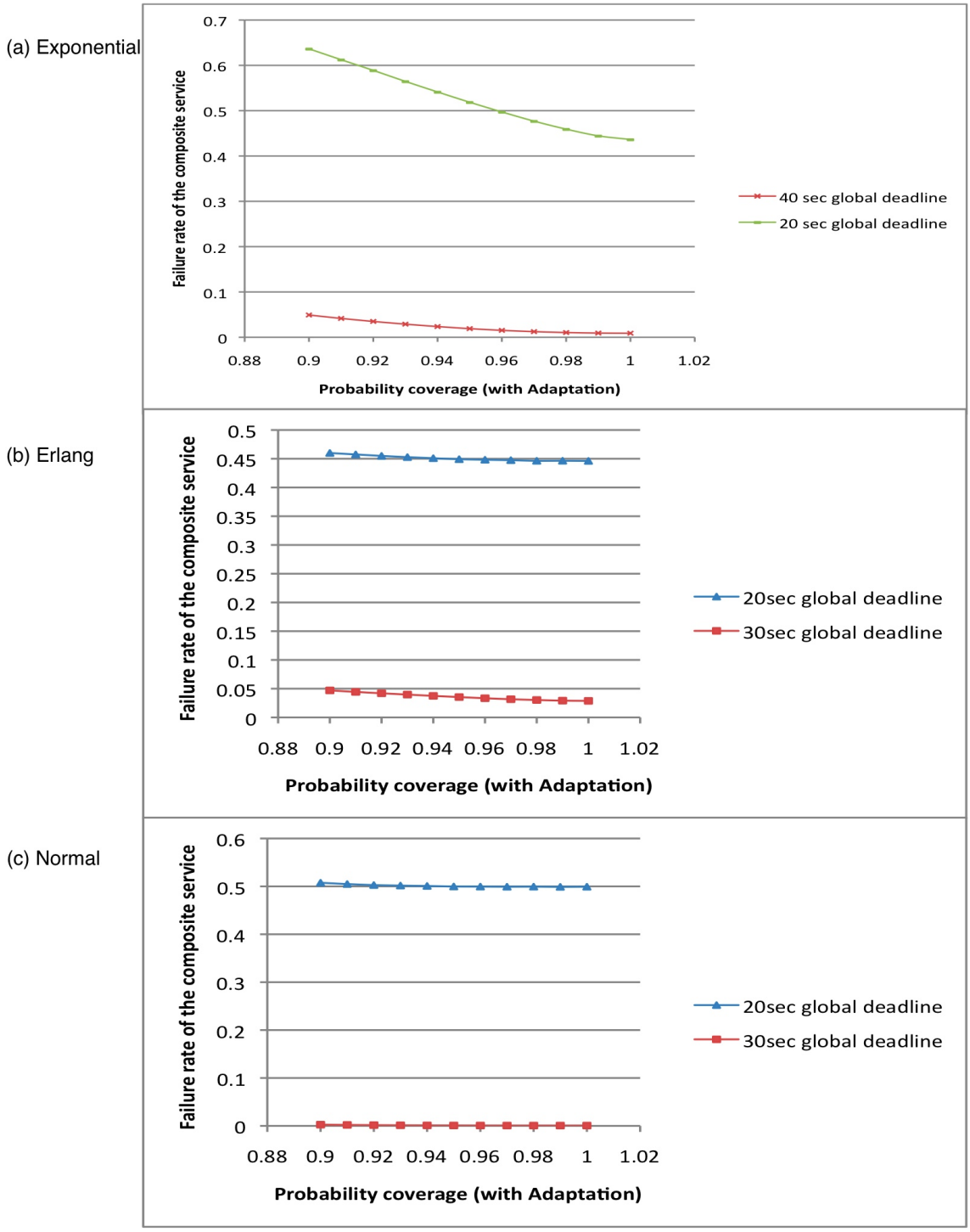


Figure 5.10 Failure rates for "Adaptation".

CHAPTER 6. FAULT-RESILIENT UBIQUITOUS SERVICE COMPOSITION

The popularity of Web Services extends to the domain of pervasive computing, as its characteristics of loose-coupling, stateless, and platform-independence make it an ideal candidate for integrating pervasive devices. While the semantics of SOA are being standardized, its use in pervasive computing is the subject of extensive research and experimentation. In fact, SOA-based pervasive computing systems are fast becoming a reality with the introduction of technology such as the Atlas Platform (56).

However, in spite of all the promises offered by favorable characteristics of SOA for coping with dynamic and heterogeneous environments, one should not forget that underneath all the nice wrappings of highly reliable and self-integrating services, the actual data sources are mass-deployed low-end sensors that are poor in terms of resources available, and they are inherently unreliable, both because of the large number of entities deployed and the common choice of employing low-cost components with few guarantees for their quality. For pervasive services to work properly and reliably, mechanisms need to be in place to improve their availability and assess the quality of their data so that necessary adjustments can be made.

Our proposed solution for building fault-resilient pervasive computing systems consists of two parts (112) (114). The first part is the Virtual Sensor framework (10) which improves the availability of basic component services. The second part consists of an architecture for performing service composition that can efficiently model, monitor and re-plan this process. In this architecture, WS-Pro (110)(109)(111), the probe-based web service composition mechanism, is adjusted to support the Abstract Service Composition Template (ASCT), a template-based service composition scheme for providing generic solutions for high-performance pervasive ser-

vice composition.

To create a comprehensive solution, these two parts have to work hand-in-hand during the entire life cycle of pervasive services. During the design stage, programmers examine the functional requirements and the type of physical sensors available to create virtual sensor services, which can then be used to match against the specifications in ASCT. During the execution stage, the compensation provided by virtual sensors provides the first line of defense against sensor failures. However, if the failures are widespread or occur within service components of higher abstraction, the WS-Pro/ASCT mechanism kicks in to identify replacement services for the failed ones.

6.1 Efficient Pervasive Service Composition

Service composition has been well studied by the web service community, and the widespread adoption of SOA in pervasive computing inspires researchers to examine whether the techniques designed for web services are equally applicable to pervasive services. Subtle but critical differences exist between the two. For instance, in web service composition it is assumed that the underlying Internet infrastructure is universal so that services can always be discovered and composed regardless of the differences in platform or communication medium used. The pervasive services, however, are tightly bound to heterogeneous hardware platforms and communication mediums, making their composition different. For example, the discovery of Bluetooth services is limited by the range of the Bluetooth device. This limitation imposes additional challenges in composition of pervasive services.

Hummel identifies fault-tolerance as a crucial issue in pervasive services (49) and points out the importance of pervasive services being able to react to dynamic changes in time. Our experience in the Gator Tech Smart House also agrees with this assessment. Different smart devices are represented as collaborating services in SOA. However, just because they work properly in collaboration does not guarantee satisfactory service to users; they may fail to deliver in time. In any typical pervasive computing environment such as a smart home, the concern for safety and security are very real and the services addressing these issues have to

be delivered promptly. In addition, users usually expect such environments to respond in a reasonably short period of time. Therefore timely delivery of services is critical. By optimizing the reconfiguration and re-composition of pervasive services, we improve the fault-tolerance as well as user satisfaction.

6.1.1 Classification of basic pervasive services

Pervasive services can be categorized into three types of abstract service elements based on their functionalities and the roles they play. As depicted in Figure 6.1, a typical end-to-end service consists of a context provider, context processor and information deliverer.

A context provider is an abstract service element that retrieves context-specific data from sensors. In other words, a context provider is a wrapper for input sensing components for detecting a specific context. In addition, each context provider has internal test functions for monitoring the health and data quality of its member components. For example, a weather monitor as a wrapper of real or virtual sensors can obtain the current temperature, humidity and wind velocity outside a house.

The context processor deals with data from context providers or a database and produces the meaningful context-based information for information deliverers. A context processor may be required to be connected with a particular set of context providers. For example, a personal scheduler can retrieve timetables from a database based on location, time and weather to provide the customized travel information.

The information deliverer is also a wrapper of a specific hardware component, such as a monitor, a printer, or an emergency alarm, used to present the information generated by the context processor. An information deliverer includes a transcoding feature that transforms the information created by the context processor into a more appropriate format. For example, a printing deliverer creates a PDF file based on the information fed from context processors and sends it to a printer.

There are two kinds of paths in the composition template: the paths between context providers or information deliverers and context processors (in the form of software services),

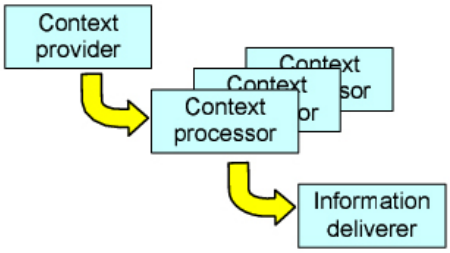


Figure 6.1 Composition of a typical end-to-end pervasive service.

and the paths among multiple software context processor services. The first path is usually performance critical and more susceptible to changes and failures. Although context processors can themselves be end-to-end services, hierarchically composed with nested context processor and context providers or information deliverers and still be highly related to the hardware devices, this association is considered indirect. Therefore we can still pinpoint the critical path as the path within the nested service provider between the component context providers or information deliverers and its directly linked context processor.

6.1.2 Performance engineering of pervasive service composition

We observe that context providers and information deliverers encounter more performance problems than context processors. The problem may exist either in the hardware layer (such as device connectivity failure) or the service layer (protocols, service failures, etc.). This problem creates a need for techniques to monitor, verify and improve system performance. It is very important to monitor both the link between services bounded by physical devices and context processors, as well as the link between the physical devices and their corresponding services, which can be either a context provider or an information deliverer.

The first link can be monitored by the pervasive service composer using probing techniques. The second link cannot be checked by the monitoring system, because not all hardware components are capable of reporting their own status. Even those devices capable of self-reporting incur high overhead to transmit data regularly. As a result, the probe has to send test messages to locate problematic hardware components. An alternative is to embed a checking mechanism when wrapping the device in a corresponding singleton service so that the health of the devices

can be monitored.

Run-time monitoring is the best solution in assessing properties of services, when traditional verification techniques are insufficient as shown in (5). Therefore, mechanisms to monitor the healthiness of pervasive equipment are highly demanded.

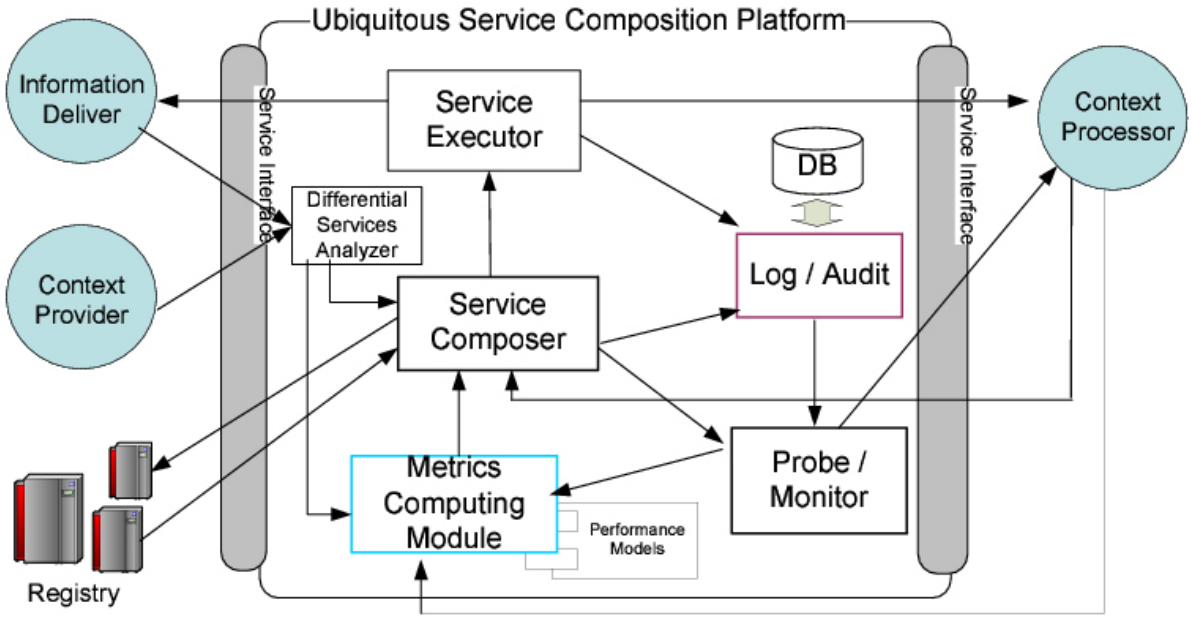


Figure 6.2 Overview of pervasive service composition

6.2 WS-Pro: A Service Composition Engine

WS-Pro was developed to model and optimize system performance in standard web service infrastructure. A testing-based probe was used to actively collect run-time performance status of component services. It supports runtime service composition and composition re-planning. A technique based on Petri net is used to model the stochastic property of service composition. This technique helps to select the best candidate services which will optimize the performance of the composite service and makes automatic verification of the composition possible. A composition re-planning algorithm was designed based on Petri net truncation. The architecture of applying WS-Pro in pervasive service composition is illustrated in Figure 6.2.

6.3 Abstract Service Composition Template (ASCT)

An Abstract Service Composition Template (ASCT) is used to describe the sequential process of critical functions to deliver a composite service. A similar approach using service templates is shown in (113). An ASCT consists of a critical flow of a service using abstract service elements that perform critical functions. Based on the abstract service elements in an ASCT, composite services can be materialized by searching and selecting appropriate actual service instances such as a context provider with assorted virtual sensors. The authors implemented a composition engine, evaluated its performance in terms of processing time and memory usage, and discussed the suitable size of the categories and the number of service elements in each category. The novelty of this approach is the introduction of an abstract layer for describing composite services and the support for their automatic composition. As a result, scenarios of a composite service can be easily described without considering strict and detailed interface description of the intended composite services. Furthermore, this makes it possible for users to create their own services or customize existing ones.

6.4 WS-Pro with ASCT Approach

As described earlier, WS-Pro was originally designed for dynamic web service composition. However, it does not adapt well to pervasive services because of two major reasons. First, dynamic composition is important in web services because there is always a large pool of candidate component services. With pervasive computing, however, the options are often much more limited in terms of functionality, making WS-Pro unsuitable for service composition. Second, all devices in pervasive computing, even low-level ones such as physical sensors, are represented as atomic services, resulting in a huge number of nodes that need to be modeled. The Generalized Stochastic Petri Net(GSPN), which is used in WS-Pro, does not scale well and hence, cannot serve as an appropriate modelling tool.

To address the first problem, we introduced the notion of ASCT. By using abstract service elements instead of well defined Web Services Definition Language (WSDL) interfaces, different devices with similar functions (for example, different presentation devices such as monitor,

printer, speaker, etc.) can be discovered and composed using the same service discovery queries. This approach also allows similar services to be considered as alternative candidates. Therefore, ASCT eliminates the problem of insufficient candidate pool at the actual service instance level when applying WS-Pro in pervasive computing environments. In addition, an ASCT can further reduce the size of the Petri Net that models the whole service composition.

The second problem can be mitigated by replacing GSPN with a Finite Population Queuing System Petri Net (FPQSPN) (15). FPQSPN extends the Petri Net by introducing "shared places and transitions," hence greatly reducing its size. Original GSPN only uses exponential distribution on transitions in order to preserve time independence. However, FPQSPN allows users to use other probability distributions which extend the GSPN's stochastic modelling capability. A Finite Population Queuing Systems Petri Net (FPQSPN) is formally defined using a 9-tuple $(P, T, Pre, Post, M_0, s_o, t, tt, k)$ such that:

- P is a finite and non-empty set of places
- T is a finite and non-empty set of transitions
- Pre is an input function, called precondition matrix of size $(|P|, |T|)$
- Post is an output function, called post-condition matrix of size $(|P|, |T|)$
- $M_0 : P(R)\{1, 2, 3, ...\}$ is an initial marking
- $t : T(R)$ $t \in R+$is the time associated to transition
- $tt : T(R)$ tt is the type of time associated to transition
- $s_o : T_i : T_i \in T, P_i : P_i \in P(R)$ $\{0, 1\}$ determines, whether the place or transition is shared
- $k : k \in N$ is number of customers (in terms of queuing systems, the size of population)

Therefore, our approach uses abstract descriptions to represent scenarios of the intended composite services. In addition to the informal approach for ASCT, we support the use of FPQSPN to describe a flow of a composite service in the composition architecture. This gives us

a concrete mathematical model to analyze the performance of services running on the proposed architecture. Accordingly, ASCTs are transformed into basic Petri Nets for verification of the semantic correctness of the composed service. Once the ASCT is realized by selecting appropriate actual service instances, we extend this basic Petri Net to FPQSPN based on their properties. A FPQSPN can efficiently depict a sub-diagram representing repeated processes by folding corresponding places or transitions into a non-shared one. We can effectively evaluate the service scenarios using multiple instances of an identical service. As discussed before, the three distinctive categories of basic pervasive services, namely context providers, context processors and information deliverers, are the necessary components of an end-to-end composite service. Therefore, an ASCT in Figure 6.3 should involve at least three different abstract service elements, with each category contributing at least one element in order to support an end-to-end pervasive service.

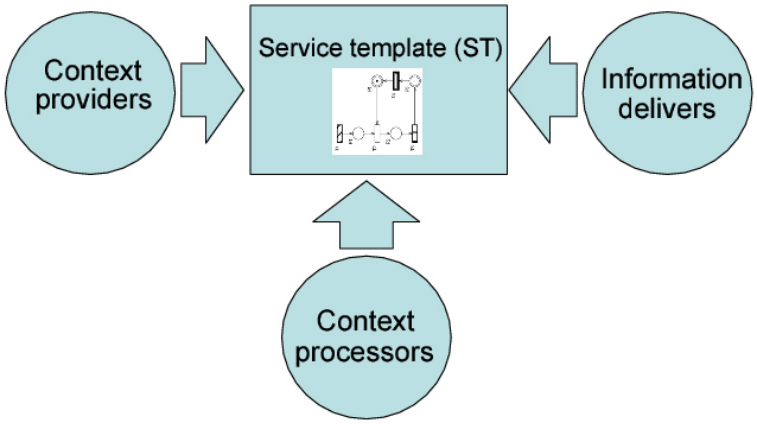


Figure 6.3 Pervasive service composition via ASCT

6.5 Enhancement in Efficient Adaptability using WS-Pro with ASCT

To demonstrate how our WS-Pro/ASCT approach enhances the efficiency in adaptation, let us consider the example of *FireHazard* described in Chapter 4.

6.5.1 ASCT for the mission critical service

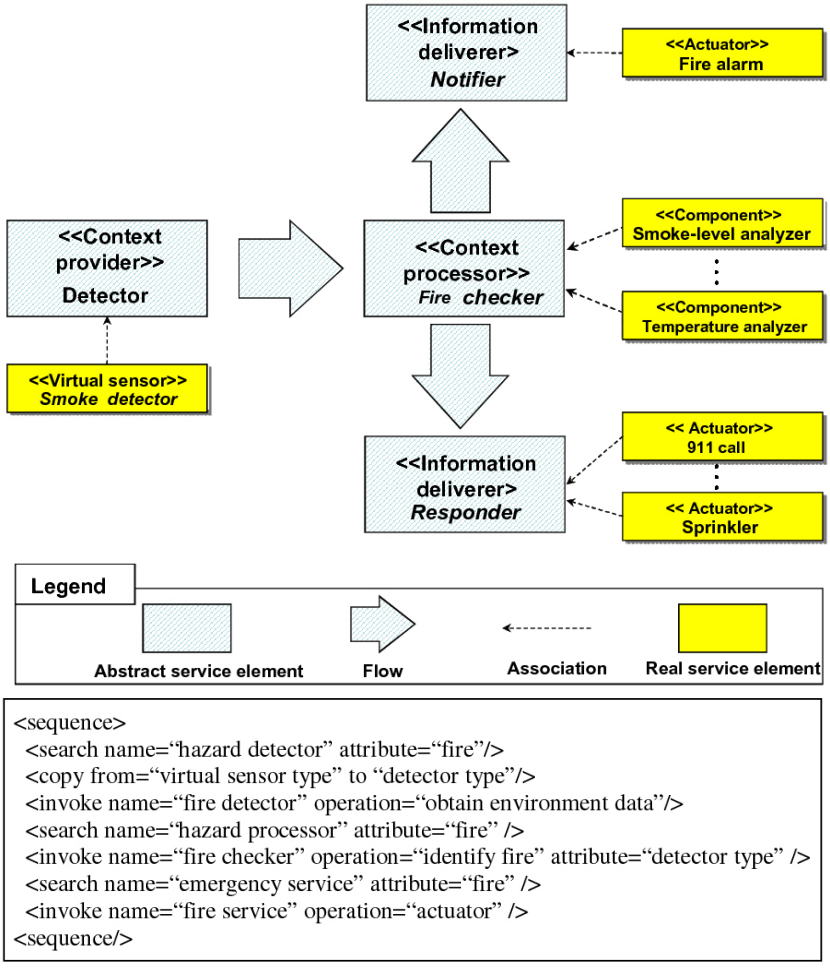
In Figure 6.4, an ASCT for an emergency fire management service is associated with a set of actual service instances. Here, the service process for the ASCT consists of abstract service elements denoted with dashed filled boxes and flows denoted with dashed filled arrows. Once appropriate real services denoted with a box are selected by WS-Pro/ASCT, they are associated with an abstract service element. These associations are denoted using shaded arrows.

According to the scenario, the smoke detectors are quickly knocked out of action. The probe in WS-Pro/ASCT captures this exception and notifies the service composer to perform service re-planning. In this case, WS-Pro/ASCT does not need to create a new ASCT. Instead, based on the current service process of the ASCT shown in Figure 6.4, WS-Pro/ASCT searches alternative actual service instances which are still alive, and associates them with the abstract service elements. For example, the smoke detector is replaced with a derived virtual sensor composed of a temperature sensor and a chemical sensor. Similarly, the abstract service element "Responder" is re-associated with an automatic 911 caller. During this process, WS-Pro/ASCT considers performance data including reliability and availability.

6.5.2 Petri Net model for mission critical service in WS-Pro/ASCT

In order to provide timely composition, the Petri Net in our WS-Pro/ASCT approach is based on the FPQSPN model. Figure 6.5 shows the FPQSPN derived from the ASCT illustrated in Figure 6.4. The object with a double line border represents a shared object such as virtual sensors while the one with a single line border depicts a non-shared object such as a unique software component.

After a FPQSPN model is generated by transforming an ASCT, it can be used to measure the performance of a working mission-critical service or evaluate the performance of a new service composed based on ASCT. Note that the real data related to the t and tt tuples in the FPQSPN model are obtained from the actual service instances associated at present. Multiple instances of the context provider are presented as a non-shared object with the number of the




```
<sequence>
  <search name="hazard detector" attribute="fire"/>
  <copy from="virtual sensor type" to "detector type"/>
  <invoke name="fire detector" operation="obtain environment data"/>
  <search name="hazard processor" attribute="fire" />
  <invoke name="fire checker" operation="identify fire" attribute="detector type" />
  <search name="emergency service" attribute="fire" />
  <invoke name="fire service" operation="actuator" />
<sequence/>
```

Figure 6.4 The ASCT for an emergency fire management service

service elements represented as k in the FPQSPN model. For this measurement for evaluation purposes, a simulator such as StpnPlay (16) is used. However, we plan to have an integrated evaluation module of FPQSPN as part of the metrics computing module in WS-Pro/ASCT.

6.6 Putting It All Together: A Comprehensive Solution for Fault-resiliency

We present the overall system architecture in Figure 6.6 which provides a comprehensive solution for fault-resiliency, by bringing virtual sensors and WS-Pro/ASCT together.

The inclusion of virtual sensors and WS-Pro/ASCT in a pervasive computing system al-

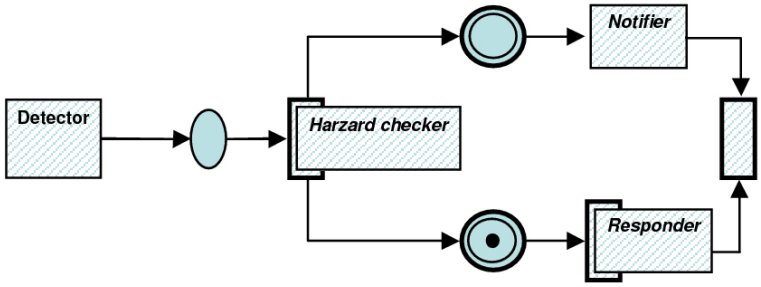


Figure 6.5 A FPQSPN for the ASCT in Figure 6.4

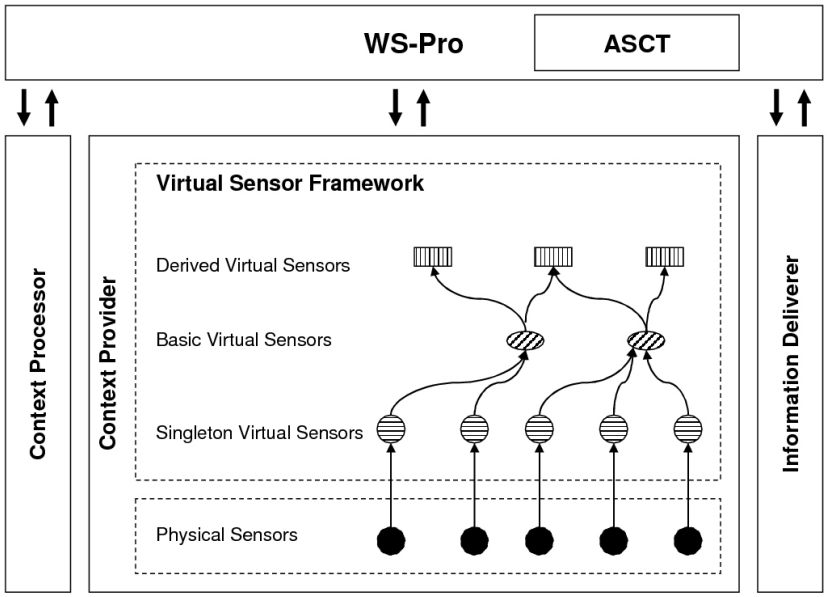


Figure 6.6 System architecture

lows it to continue functioning properly and degrading gracefully in face of sensor failures. Virtual sensors enable this by exploiting explicit redundancy (replicas) or indirect redundancy (correlated sensors) to compensate for the loss of data. However, when the system experiences extensive failures or becomes unstable, WS-Pro/ASCT kicks in and exploits redundancy at a higher level in the form of semantically equivalent services to stablize the system. Even though both share the same objective of enhancing availability, each works at different levels and employs independent mechanisms, and their strengths complement each other. To provide a comprehensive solution to address the issue of fault-resiliency, it is crucial to ensure a logical and smooth integration at various stages in the life cycle of the system.

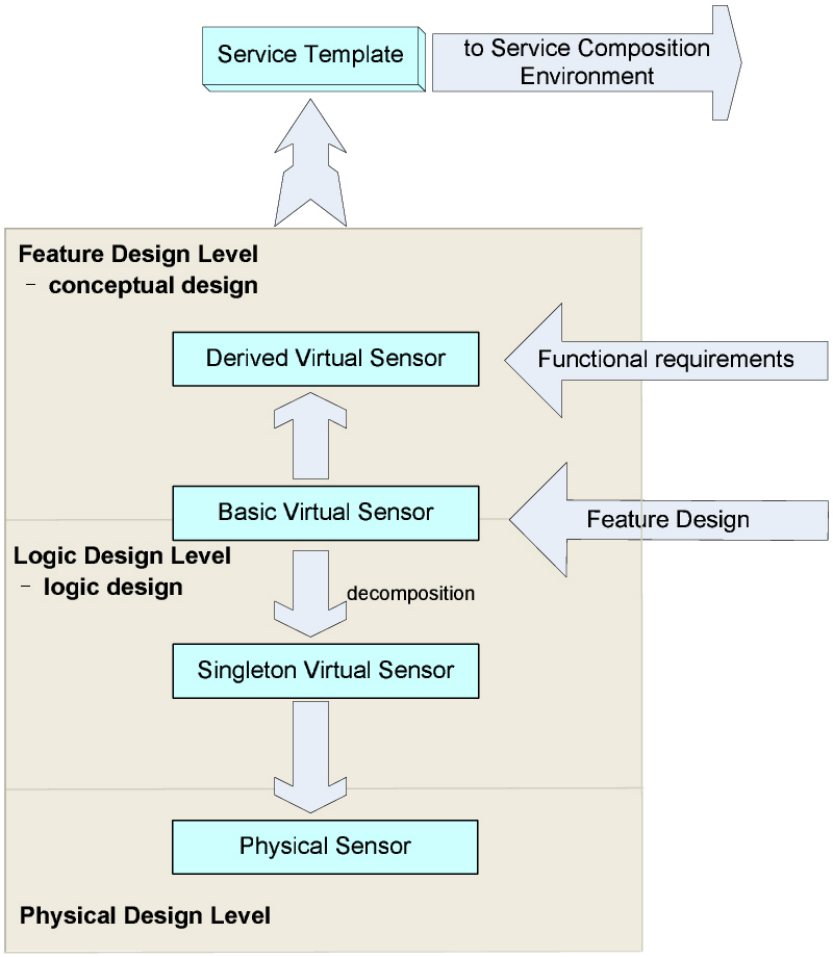


Figure 6.7 Design process for integrating virtual sensors into service templates

The aim of the virtual sensor is to provide a robust and high quality data source. The designers of a system look at the features of services in a system, and decide which sensors to deploy, and the required level of fault-resiliency as shown in 6.7. The feature design dictates what kinds of basic virtual sensor needed to be implemented, which include attributes such as the number, quality and spatial distribution of singleton virtual sensors, as well as the aggregation algorithm to be used. This decomposition process gives a blueprint of which physical sensors to use, as well as where and how they should be deployed. On the other front, the functional requirements of services would justify the conceptual design and composition of multiple basic virtual sensors into a derived virtual sensor. Some of the reasons to design derived

virtual sensors include dimensions which do not have means for direct measurement, the need for more comprehensive and abstract contextual information than raw readings, and the frequent reuse of certain aggregated data and information. As all virtual sensors are implemented as services, any of the singleton, basic, or derived virtual sensors can be a candidate in the service composition process. WS-Pro/ASCT service composition mechanism can match and choose these virtual sensor services based on various criteria, for instance, the fault-tolerance requirement, or whether raw data is preferred over comprehensive contexts.

These two pieces of the puzzle also work closely during runtime operation. Each virtual sensor service by default constantly monitors its member virtual sensors and tries to compensate should any of them fail. In the case involving wide-spread sensor failures or malfunctioning of hard-to-compensate sensors, the quality of virtual sensor, might fall below certain predefined threshold. Should such situation occur, the virtual sensor immediately notifies the WS-Pro module and requests for a service replanning. WS-Pro works in tandem with the ASCT to replan the services utilizing the failed virtual sensor, and search for a replacement virtual sensor service to prevent interruption and breakdown of the overall service.

CHAPTER 7. SUMMARY AND DISCUSSION

7.1 Summary

With the prevalence of software on almost every aspect of our social life, the complexity and size of software systems have profoundly increased. Practitioners have been seeking a highly effective software design paradigm that can realize rapid software development, adapt to fast changing business needs, and more importantly, seamlessly integrate legacy systems with other legacy systems and new developments. Responding to this demand, SOA was proposed as a critical concept to create software systems using available applications and to facilitate software integration.

Recent great advances in XML inspired researchers to build an interoperability stack, called Web Services Architecture (9), based on XML technology. This stack contains standards and specifications from business level to the lower implementation level. This stack is purely XML-based that provides developers great flexibility and interoperability. Hence, Web Services is widely accepted to be the best way to implement SOA and is becoming the dominant software design paradigm in industry.

While Web Services research is still in its in-mature stage and its implementation details are still to finalize, we should start to consider the performance engineering issue. Industry has learned through numerous hard lessons (107) the significance of well founded and well executed performance analysis plans. We identified three key performance engineering issues in the Web Services framework, performance modelling, performance-based service composition, and performance adaptation. To address these issues, this dissertation proposed a comprehensive performance management approach, called WS-Pro.

We provide support for performance prediction. We designed a transformation to generate

Petri net from business process described in WS-BPEL. Through the generated Petri nets, we can verify system properties and predict performance to certain extent. The prediction can help practitioners evaluate their design. This early feedback can help mitigate software cost and business lost.

We also addressed the problem of performance adaptation in Web Services. Web Services paradigm is a heterogenous distributed computing that can be dynamically configured. Services are all autonomous applications which are beyond software developers' control. We need to make sure the composite service will not stop functioning due to single failures. Considering from performance aspect, we not only need to keep the service functioning, but also try to deliver service on time. It requires a mechanism that can respond to exceptions (failure or delay) quickly. Unfortunately, very limited research has been conducted to solve these problems. Our adaptation mechanism can replace failed services or switch to other execution paths if there are alternative ones. We also focused on improving the performance of our adaptation mechanism to provide timely support.

As an extension, WS-Pro was extended to fit into the ubiquitous environment. We proposed an abstract service description template, called Abstract Service Composition Template (ASCT), to tackle the obstacle we faced – the lack of available functional components. This extension work is designed to enhance fault-resilience combining with other platform-related techniques.

7.2 Contribution

The contribution of this research contains:

- **A general framework to address performance engineering issues in Web Services.**

  We designed a service composition framework that guides service selection and dynamically re-configures the composition based on several theoretically sound algorithms.

- **A transformation from WS-BPEL to Petri net**

Our first research objective was to provide analytical support to service composition procedure. WS-Pro is based on the formalism of transforming WS-BPEL to Petri net. We transform business processes written in WS-BPEL into analytic model represented in Petri nets. We also proved the soundness of the transformation. The results of the transformation can be used for analytical purposes.

- **A two-phase algorithm to compute optimal execution plan**

  We designed algorithms to compute optimal execution plans based on the Petri nets. The WS-Pro is aimed to optimize performance of service composition to address designer's concerns in two aspects: performance of the delivered service and performance of the composition procedure. We are particularly concerned about the latter one as it is often neglected in existing works. The algorithms we designed contain off-line part and runtime part. We were able to minimize the runtime computation but had to compromise a little in terms of the total computation (i.e., the extra steps of compression and loop removal). Our experiments generated promising results to show the efficiency of our design.

- **A performance-based adaptation algorithm**

  We cannot prevent failure because services' behavior is highly unpredictable. Instead, we provide the runtime adaptation mechanism to handle performance exceptions. In the adaptation mechanism, we considered business alliances and financial cost-efficiency. We evaluate our approach using both real cases and simulation. The experiments generated promising results that had demonstrated the effectiveness of WS-Pro.

- **Extension to ubiquitous computing**

  Ubiquitous (pervasive) computing is a fast growing area which quickly adopts the service-oriented concept because of its power of composition. However, in a ubiquitous environment service composition often suffers from slow integration of a massive number of sensors and limited alternatives at the application service level. The latter problem restricts the use of our performance adaptation mechanism defined in the Web Services setting. To overcome this, we defined the Abstract Service Composition Template (ASCT) (112)

(114). Through ASCT, a service pool can scale up using services of the same meta-level functionalities.

7.3 Future Work

WS-Pro can be improved in different ways. We outline several major directions as follows.

- **A tool to support automatic transformation**

  "Dynamic" is a key word in Web Services because the framework is designed for autonomous computing. Service composition environment should be able to automatically discover and select component services, automatically compose and re-configure them based on pre-specified documents (i.e. WSDL and BPEL description). In order to fully automate WS-Pro, we need a tool to perform the transformation task. It will allow non-experts to fully use the functionalities provided by WS-Pro without understanding the underlying theoretical background.

- **A powerful tool to generate reachability graph for Petri nets**

  Currently we use PIPE2 to generate reachability graphs. PIPE2 is favored by many researchers because of its graphic interface. It also has various stochastic analysis features that are very useful for performance study. However, PIPE2 has limited computational power - it can only compute Petri nets with fewer than 32 places (with a few loops and branches). Moreover, we could not use PIPE2 through back-end. Currently PIPE2 is separate from other components including compressor, loop removal program, etc.

- **Performance data mining**

  Performance data provides important references to select best services. The data are mainly reasoned out from historical data. We have tried different statistical approaches, such as F-test and ANOVA - Bonferroni, to analyze response time and confidence interval. In empirical studies, re-sampling and non-parametric statistic estimation techniques can be used if the available data is not adequate. When large amount of data are available, more powerful data mining techniques should be considered.

- **Web Services testing**

  A test component will improve the healthiness of WS-Pro. There are situations where historical data and SLA requirements are not reliable. A fast probing technique will allow WS-Pro to understand the real status of services and take corresponding actions.

  In the Web Services framework, we need to test the composite business logic without knowing implementation details of each component. This testing process needs to be adaptive to dynamic service binding and dynamic service composition. Therefore, the test suit needs to have good re-configurability to test different bindings of component services. It can also involve a search-based algorithm to select bindings because we cannot exhaustively test all the possible system configurations.

  Furthermore, the connection between the local monitoring component and testing component should be reinforced in many ways. The data collected through monitoring can be use to guide the generation of test data. Pre and post conditions can be derived through logic reasoning on the historical data. These conditions are useful when generating test logic.

- **Empirical study**

  Though our research prototype demonstrates promising results, an empirical study is still highly desired. This requires a Web Services environment as test bed. Smart Home being developed in both Iowa State University (91) (95) and University of Florida (47) will be such environments.

- **A new software design in service-oriented ubiquitous computing**

  When we put our WS-Pro framework and the virtual sensor network together (Chapter 6), we encountered integration problem. There is some overlapping between the two frameworks. However, this overlapping does not merge smoothly. We then proposed the feature-based design – starting the service design at the feature level and then going upward for conceptual design or going downward for business logic design. This design

paradigm needs to be refined from software engineering's aspect. It will advance the research of service-oriented design in pervasive environment.

- **Extension to a comprehensive Qos-based framework**

  WS-Pro is a performance-based service composition framework. It can be expanded to the general QoS research by including more quality attributes. This requires a multi-dimensional model. Such a model should have a mechanism to compute a total quality value based on different prioritization. We will investigate the possibility of using Bayesian network as such a model.

APPENDIX A. SYNTAX OF WS-BPEL

Notation:

O(x) ::= empty | x

#(x) ::= any number of x

P(x) ::= x #(x)

U(x,y) ::= any order of x and y

All the basic activities can be in the "<activity attributes />" form when there is no elements included. For example, the invoke activity may be defined as: "<invoke" invoke-attributes "/>".

WS-BPEL *::= U(O(import), O(documentation), O(partnerlink), O(variables), O(activities), O(handler), O(extension), O(messageExchanges), O(correlationSets))*

Activities *::= P(basic-activities | structured-activities)*

Basic-activities *::= Invoke | Receive | Reply | Assign | Throw | Wait | Empty | Extension activity | Exit | Rethrow*

Structured-activities *::= Sequence | If | While | RepeatUntil | Pick | Flow | ForEach | Compensate | CompensateScope*

standard-attributes *::= U(O(name), O(suppressJoinFailure))*

standard-elements *::= U(O(targets), O(sources))*

Invoke *::= "<invoke" invoke-attributes ">" invoke-elements "</invoke>"*

invoke-attributes *::= U(standard-attributes, partnerLink, O(portType), operation, O(inputVariable), O(outputVariable))*

invoke-elements *::= U(standard-elements, O(correlations), O(catch), O(compensationHandler), O(toParts), O(fromParts))*

Receive *::= "<receive" receive-attributes">" U(standard-elements, O(correlations), O(fromParts)) "</receive>"*

receive-attributes *::= U(standard-attributes, partnerLink, O(portType), operation, O(variable), O(createInstance), O(messageExchange))*

Reply *::= "<reply" reply-attributes ">"U(standard-elements, O(correlations), O(toParts)) "</reply>"*

reply-attributes *::= U(standard-attributes, partnerLink, O(portType), operation, O(variable), O(faultName), O(messageExchange))*

Assign *::= "<assign" O(validate) standard-attributes ">" U(standard-elements, P(copy | O(extensionAssignOperation))) "</Assign>"*

copy *::= "<copy>" from to "</copy>" (the definition of "from" and "to" are skipped here)*

Throw *::= "<throw" throw-attributes ">" standard-elements "</Throw>"*

throw-attributes *::= U(standard-attributes, O(faultName), O(faultVariable))*

Wait *::= "<wait" standard-attributes ">" U(standard-elements, O(for) | O(until)) "</wait>"*

Empty *::= "<empty" standard-attributes ">" standard-elements "</empty>"*

ExtensionActivity *::= "<extensionActivity" #(anyElementQName) "</extensionActivity>"*

Exit *::= "<exit" standard-attributes ">" standard-elements "</exit>"*

Rethrow *::= "<rethrow" standard-attributes ">" standard-elements "</rethrow>"*

Sequence *::= "<sequence" standard-attributes ">" U(standard-elements, Activities)*

"</sequence>"

If *::= "<if" standard-attributes">"standard-elements condition Activities #(elseif) O(else) "</if>"*

While *::= "<while" standard-attributes ">" standard-elements condition Activities "</while>"*

RepeatUntil *::= "<repeatUntil" standard-attributes ">" standard-elements Activities condition "</repeatUntil >"*

Pick *::= "<pick" pick-attributes ">"U(standard-elements, P(onMessage), O(onAlarm)) "</pick>"*

pick-attributes *::= U(O(createInstance), standard-attributes)*

onMessage *::= "<onMessage" message-attributes ">" O(correlations) O(fromParts) Activities "</onMessage>"*

onAlarm *::= "<onAlarm> (" O(for | until) "</onAlarm>"*

flow *::= "<flow" standard-attributes ">" U(standard-elements, O(links), Activities) "</flow>"*

forEach *::= "<forEach" foreach-attributes ">"U(standard-elements, startCounterValue, finalCounterValue, O(completionCondition)) Scope "</forEach>"*

foreach-attributes *::= U(counterName, parallel, standard-attributes)*

scope *::= "<scope" scope-attributes ">" scope-elements "</scope>"*

scope-attributes *::= U(standard-attributes, O(isolated), exitOnStandardFault)*

scope-elements *::= U(standard-elements, O(partnerLink), O(messageExchanges),*

O(variables), O(correlationSets), #(handlers), Activities)

compensateScope *::= "<compensateScope" U(target, standard-attributes) ">*
"standard-elements "</compensateScope>"

compensate *::= "<compensate" standard-attributes ">"standard-elements*
"</compensate>"

APPENDIX B. BUSINESS PROCESS DESCRIPTION OF THE SEEMOVIE SERVICE

```
<?xml version = "1.0" encoding = "UTF-8" ?>
<!-- <process
name="SeeMovie" suppressJoinFailure="yes"
targetNamespace="http://SeeMovie"
         xmlns="http://schemas.xmlsoap.org/ws/2003/03/business-process/"
         xmlns:bpws="http://schemas.xmlsoap.org/ws/2003/03/business-process/"
         xmlns:ns1="http://services.otn.com"
         xmlns:xsd="http://www.w3.org/2001/XMLSchema">
    <!--
      ////////////////////////////////////////////////////////////////////////////
        PARTNERLINKS
        List of services participating in this BPEL process
      ////////////////////////////////////////////////////////////////////////////
    -->
    <partnerLinks>
        <!--
      The 'client' role represents the requester of this service. It is
      used for callback. The location and correlation information associated
      with the client role are automatically set using WS-Addressing.
    -->
        <partnerLink name="client" partnerLinkType="client:SeeMovie"
                     myRole="SeeMovieProvider" partnerRole="SeeMovieRequester"/>
        <partnerLink name="MovieScheduleService"
                     partnerLinkType="ns1:MovieScheduleService"
```

```
                partnerRole="MovieScheduleServiceProvider"/>
    <partnerLink name="TicketService" partnerLinkType="ns1:TicketService"
                partnerRole="TicketServiceProvider"/>
    <partnerLink name="BusScheduleService"
                partnerRole="BusScheduleServiceProvider"
                partnerLinkType="ns1:BusScheduleService"/>
    <partnerLink name="BusFareService" partnerRole="BusFareServiceProvider"
                partnerLinkType="ns1:BusFareService"/>
</partnerLinks>
<!--
  ////////////////////////////////////////////////////////////////////////////
    VARIABLES
    List of messages and XML documents used within this BPEL process
  ////////////////////////////////////////////////////////////////////////////
-->
<variables>
    <!-- Reference to the message passed as input during initiation -->
    <variable name="inputVariable"
              messageType="client:SeeMovieRequestMessage"/>
    <!-- Reference to the message that will be sent back to the requester during
        callback -->
    <variable name="outputVariable"
              messageType="client:SeeMovieResponseMessage"/>
    <variable name="Invoke_checkSchedule_schedule_InputVariable"
              messageType="ns1:MovieScheduleServiceRequestMessage"/>
    <variable name="Invoke_checkSchedule_schedule_OutputVariable"
              messageType="ns1:MovieScheduleServiceResponseMessage"/>
    <variable name="Invoke_BusSchedule_busschedule_InputVariable"
              messageType="ns1:BusScheduleServiceRequestMessage"/>
    <variable name="Invoke_BusSchedule_busschedule_OutputVariable"
              messageType="ns1:BusScheduleServiceResponseMessage"/>
    <variable name="Invoke_BusFare_fare_InputVariable"
```

```
            messageType="ns1:BusFareServiceRequestMessage"/>
    <variable name="Invoke_BusFare_fare_OutputVariable"
            messageType="ns1:BusFareServiceResponseMessage"/>
    <variable name="Invoke_Ticket_process_InputVariable"
            messageType="ns1:TicketServiceRequestMessage"/>
    <variable name="Invoke_Ticket_process_OutputVariable"
            messageType="ns1:TicketServiceResponseMessage"/>
    <variable name="CancelTrip_InputVariable"
            messageType="ns1:CancelMessage"/>
</variables>
<eventHandlers>
    <onMessage portType="client:CancelMovie" operation="cancel"
             partnerLink="client" variable="CancelTrip_InputVariable">
        <sequence name="Sequence_3">
            <compensate name="Compensate_CancelTrip"/>
            <terminate name="Exit"/>
        </sequence>
    </onMessage>
</eventHandlers>
<!--
  ////////////////////////////////////////////////////////////////////////////
   ORCHESTRATION LOGIC
   Set of activities coordinating the flow of messages across the
   services integrated within this business process
  ////////////////////////////////////////////////////////////////////////////
-->
<sequence name="main">
    <!-- Receive input from requestor. (Note: This maps to operation defined in
        SeeMovie.wsdl) -->
    <receive name="receiveInput" partnerLink="client"
            portType="client:SeeMovie" operation="initiate"
            variable="inputVariable" createInstance="yes"/>
```

```
<!--
  Asynchronous callback to the requester. (Note: the callback location and
  correlation id is transparently handled using WS-addressing.)
-->
<scope name="Scope_seemovie">
    <flow name="Flow_seemovie">
        <links>
            <link name="busToticket"/>
            <link name="ticketTobus"/>
        </links>
        <sequence name="Sequence_1">
            <assign name="Assign_1">
                <copy>
                   <from variable="inputVariable" part="payload"
                       query="/client:SeeMovieProcessRequest/client:input"/>
                   <to variable="Invoke_checkSchedule_schedule_InputVariable"
                       part="payload" query="/ns1:title"/>
                </copy>
            </assign>
            <invoke name="Invoke_MovieSchedule"
               partnerLink="MovieScheduleService"
               portType="ns1:MovieScheduleService"
               operation="schedule"
               inputVariable="Invoke_checkSchedule_schedule_InputVariable"
               outputVariable="Invoke_checkSchedule_schedule_OutputVariable">
               <source linkName="ticketTobus"/>
            </invoke>
            <assign name="Assign_4">
              <copy>
               <from variable="Invoke_checkSchedule_schedule_OutputVariable"
                        part="payload" query="/ns1:info"/>
               <to variable="Invoke_Ticket_process_InputVariable"
```

```
                part="payload" query="/ns1:movieinfo"/>
      </copy>
    </assign>
    <invoke name="Invoke_Ticket" partnerLink="TicketService"
            portType="ns1:TicketService" operation="process"
            inputVariable="Invoke_Ticket_process_InputVariable"
            outputVariable="Invoke_Ticket_process_OutputVariable">
        <target linkName="busToticket"/>
    </invoke>
</sequence>
<sequence name="Sequence_2">
    <assign name="Assign_2">
        <copy>
          <from variable="inputVariable" part="payload"
               query="/client:SeeMovieProcessRequest/client:input"/>
          <to variable="Invoke_BusSchedule_busschedule_InputVariable"
                part="payload" query="/ns1:locations"/>
        </copy>
    </assign>
    <invoke name="Invoke_BusSchedule"
      partnerLink="BusScheduleService"
      portType="ns1:BusScheduleService"
      operation="busschedule"
      inputVariable="Invoke_BusSchedule_busschedule_InputVariable"
      outputVariable="Invoke_BusSchedule_busschedule_OutputVariable">
      <target linkName="ticketTobus"/>
    </invoke>
    <assign name="Assign_3">
      <copy>
       <from variable="Invoke_BusSchedule_busschedule_OutputVariable"
             part="payload" query="/ns1:info"/>
       <to variable="Invoke_BusFare_fare_InputVariable"
```

```
                        part="payload" query="/ns1:route"/>
                    </copy>
                </assign>
                <invoke name="Invoke_BusFare" partnerLink="BusFareService"
                        portType="ns1:BusFareService" operation="fare"
                        inputVariable="Invoke_BusFare_fare_InputVariable"
                        outputVariable="Invoke_BusFare_fare_OutputVariable">
                    <source linkName="busToticket"/>
                </invoke>
            </sequence>
        </flow>
        <invoke name="callbackClient" partnerLink="client"
            portType="client:SeeMovieCallback" operation="onResult"
            inputVariable="outputVariable"/>
    </scope>
  </sequence>
</process>
```

APPENDIX C. BUSINESS PROCESS DESCRIPTION OF THE FIREHAZARD SERVICE

```
<?xml version = "1.0" encoding = "UTF-8" ?><process
name="FireHazard" suppressJoinFailure="yes"
targetNamespace="http://xmlns.oracle.com/FireHazard"
         xmlns="http://schemas.xmlsoap.org/ws/2003/03/business-process/"
         xmlns:bpws="http://schemas.xmlsoap.org/ws/2003/03/business-process/"
         xmlns:xsd="http://www.w3.org/2001/XMLSchema"
         xmlns:ns1="http://services.otn.com"
    <!--
      ////////////////////////////////////////////////////////////////////////////
        PARTNERLINKS
        List of services participating in this BPEL process
      ////////////////////////////////////////////////////////////////////////////
    -->
    <partnerLinks>
        <!--
      The 'client' role represents the requester of this service. It is
      used for callback. The location and correlation information associated
      with the client role are automatically set using WS-Addressing.
    -->
        <partnerLink name="Smoke_detector" partnerLinkType="client:FireHazard"
                     myRole="SmokeProvider" partnerRole="SmokeRequester"/>
        <partnerLink name="Chemical_detector"
                     partnerLinkType="ns1:ChemicalService"
                     partnerRole="ChemicalServiceProvider"/>
```

```
    <partnerLink name="SprinklerService" partnerLinkType="ns1:SprinklerService"
                 partnerRole="SprinklerServiceProvider"/>
    <partnerLink name="AlarmService"
                 partnerRole="AlarmServiceProvider"
                 partnerLinkType="ns1:AlarmService"/>
    <partnerLink name="Temperature_detector"
                 partnerRole="TemperatureProvider"
                 partnerLinkType="ns1:TemperatureService"/>
    <partnerLink name="NotificationService"
                 partnerRole="NotificationServiceProvider"
                 partnerLinkType="ns2:NotificationServiceLink"/>
</partnerLinks>
<!--
  ////////////////////////////////////////////////////////////////////////
    VARIABLES
    List of messages and XML documents used within this BPEL process
  ////////////////////////////////////////////////////////////////////////
-->
<variables>
    <!-- Reference to the message passed as input during initiation -->
    <variable name="inputVariable"
              messageType="client:FireHazardRequestMessage"/>
    <!-- Reference to the message that will be sent back to the requester
          during callback -->
    <variable name="outputVariable"
              messageType="client:FireHazardResponseMessage"/>
    <variable name="Invoke_smoke_onResult_InputVariable"
              messageType="client:FireHazardResponseMessage"/>
    <variable name="Invoke_chemical_InputVariable"
              messageType="ns1:ChemicalServiceRequestMessage"/>
    <variable name="Invoke_temperature_InputVariable"
              messageType="ns1:TemperatureServiceRequestMessage"/>
```

```
    <variable name="Invoke_Alarm_InputVariable"
              messageType="ns1:AlarmServiceRequestMessage"/>
    <variable name="Invoke_sprinkler_process_InputVariable"
              messageType="SprinklerServiceRequestMessage"/>
</variables>
<!--
  ////////////////////////////////////////////////////////////////////////
   ORCHESTRATION LOGIC
   Set of activities coordinating the flow of messages across the
   services integrated within this business process
  ////////////////////////////////////////////////////////////////////////
-->
<sequence name="main">
    <!-- Receive input from requestor. (Note: This maps to operation defined
       in FireHazard.wsdl) -->
    <receive name="Receive_1" partnerLink="Smoke_detector"
             portType="client:FireHazard" operation="initiate"
             variable="inputVariable" createInstance="no"/>
    <while name="While_1"
           condition="bpws:getVariableData('inputVariable','payload','/client
           :FireHazardProcessRequest/client:input') = 0">
        <sequence name="Sequence_WHILE">
            <flow name="Flow_fire_control">
                <sequence name="Sequence_8">
                    <scope name="Voice_1">
                        <bpelx:annotation>
                            <bpelx:pattern patternName="bpelx:voice">
                            </bpelx:pattern>
                        </bpelx:annotation>
                        <variables>
                          <variable name="varNotificationReq"
                            messageType="ns2:VoiceNotificationRequest"/>
```

```
  <variable name="varNotificationResponse"
    messageType="ns2:ArrayOfResponse"/>
  <variable name="NotificationServiceFaultVariable"
    messageType=
    "ns2:NotificationServiceErrorMessage"/>
</variables>
<sequence name="Sequence_9">
    <assign name="VoiceParamsAssign">
        <copy>
            <from expression="string('')"/>
            <to variable="varNotificationReq"
                part="VoicePayload"
                query="/VoicePayload/ns2:Content
                /ns2:ContentBody"/>
        </copy>
        <copy>
            <from expression="string('text/vxml')"/>
            <to variable="varNotificationReq"
                part="VoicePayload"
                query="/VoicePayload/ns2:Content/
                ns2:MimeType"/>
        </copy>
        <copy>
            <from expression="string('43235234')"/>
            <to variable="varNotificationReq"
                part="VoicePayload"
                query="/VoicePayload/ns2:To"/>
        </copy>
    </assign>
    <invoke name="InvokeNotificationService"
            partnerLink="NotificationService"
            portType="ns2:NotificationService"
```

```
                    operation="sendVoiceNotification"
                    inputVariable="varNotificationReq"
                    outputVariable="varNotificationResponse"/>
        </sequence>
    </scope>
</sequence>
<sequence name="Sequence_7">
    <assign name="Assign_5">
      <copy>
         <from variable="Invoke_smoke_onResult_InputVariable"
              part="payload" query="/ns1:route"/>
         <to variable="Invoke_sprinkler_process_InputVariable"
              part="payload" query="/ns1:fireinfo"/>
      </copy>
    </assign>
    <invoke name="Invoke_sprinkler"
       partnerLink="SprinklerService"
       portType="ns1:SprinklerService" operation="process"
       inputVariable="Invoke_sprinkler_process_InputVariable"/>
</sequence>
<sequence name="Sequence_7">
    <assign name="Assign_4">
       <copy>
          <from variable="Invoke_smoke_onResult_InputVariable"
             part="payload"
             query="/client:FireHazardProcessResponse/
             client:result"/>
            <to variable="Invoke_Alarm_InputVariable"
                part="payload" query="/ns1:fireinfo"/>
       </copy>
    </assign>
    <invoke name="Invoke_1" partnerLink="AlarmService"
```

```
                portType="ns1:AlarmService"
                operation="process"
                inputVariable="Invoke_Alarm_InputVariable"/>
    </sequence>
</flow>
<switch name="Switch_fire_checker">
    <case condition="bpws:getVariableData('inputVariable','payload',
    '/client:FireHazardProcessRequest/client:input') =2">
        <sequence name="Sequence_5">
            <assign name="Assign_2">
                <copy>
                    <from variable="inputVariable"
                          part="payload"
                          query="/client:FireHazardProcessRequest/
                          client:input"/>
                    <to variable="Invoke_chemical_InputVariable"
                        part="payload" query="/ns1:title"/>
                </copy>
            </assign>
            <invoke name="Invoke_chemical"
                    partnerLink="Chemical_detector"
                    portType="ns1:ChemicalService"
                    operation="process"
                    inputVariable="Invoke_chemical_InputVariable"/>
        </sequence>
    </case>
    <case condition="bpws:getVariableData('inputVariable','payload',
    '/client:FireHazardProcessRequest/client:input') = 1">
        <sequence name="Sequence_4">
            <assign name="Assign_1">
               <copy>
                 <from variable="inputVariable"
```

```
                        part="payload"
                        query="/client:FireHazardProcessRequest/
                        client:input"/>
                <to variable="Invoke_smoke_onResult_InputVariable"
                        part="payload"
                        query="/client:FireHazardProcessResponse/
                        client:result"/>
            </copy>
        </assign>
        <invoke name="Invoke_smoke"
          partnerLink="Smoke_detector"
          portType="client:FireHazardCallback"
          operation="onResult"
          inputVariable="Invoke_smoke_onResult_InputVariable"/>
    </sequence>
</case>
<otherwise>
    <sequence name="Sequence_6">
        <assign name="Assign_3">
            <copy>
                <from variable="inputVariable"
                      part="payload"
                      query="/client:FireHazardProcessRequest/
                      client:input"/>
                <to variable="Invoke_temperature_InputVariable"
                    part="payload" query="/ns1:route"/>
            </copy>
        </assign>
        <invoke name="Invoke_temperature"
           partnerLink="Temperature_detector"
           portType="ns1:TemperatureService"
           operation="process"
```

```
                        inputVariable="Invoke_temperature_InputVariable"/>
                    </sequence>
                </otherwise>
            </switch>
            <assign name="Assign_onFire">
                <copy>
                   <from variable="Invoke_sprinkler_process_InputVariable"
                     part="payload" query="/ns1:fireinfo"/>
                   <to variable="outputVariable" part="payload"
                     query="/client:FireHazardProcessResponse/client:result"/>
                </copy>
            </assign>
        </sequence>
    </while>
    <!--
      Asynchronous callback to the requester. (Note: the callback location and
      correlation id is transparently handled using WS-addressing.)
    -->
  </sequence>
</process>
```

APPENDIX D. Simulation screenshot

```
********************************************************************************
TEST 0
********************************************************************************
Initializing test 0
Distance from t1 to t23: 19799.999999999996
Path: [t1, t2, t3, t5, t8, t11, t14, t17, t20, t22, t23]
============================================================
Service t22 timedout
Alter Service's new estimated time is: 19799.999999999996(PathEstimateTime)+3600(ServiceTimeout)
-1200.0(serviceAverageTime)+1400.0(AltServiceAverageTime)+0(AlgrithmTime)=23599.999999999996
totalPathLimit is 200000
Using service t22 's alter
alg time is :0
result:
[t1, t2, t3, t5, t8, t11, t14, t17, t20, t22, t23]
COMPLETE[ExecutionPath ExecutionPath0 completed]
Time cal spent: 29792
Overhead Time: 0
Stage Staget1 spent 1463
Stage Staget2 spent 425
Stage Staget3 spent 2955
Stage Staget5 spent 2274
Stage Staget8 spent 1550
Stage Staget11 spent 10858
Stage Staget14 spent 1407
```

Stage Staget17 spent 3526

Stage Staget20 spent 1377

Stage Staget22 spent 3957

Total is 29792

Test 0 completed

****

TEST 1

****

Initializing test 1

Distance from t1 to t23: 19799.999999999996

Path: [t1, t2, t3, t5, t8, t11, t14, t17, t20, t22, t23]

==========

Service t17 timedout

Alter Service's new estimated time is: 19799.999999999996(PathEstimateTime)+6600(ServiceTimeout)

-2200.0(serviceAverageTime)+2300.0(AltServiceAverageTime)+0(AlgrithmTime)=26499.999999999996

totalPathLimit is 200000

Using service t17 's alter

alg time is :0

result:

[t1, t2, t3, t5, t8, t11, t14, t17, t20, t22, t23]

COMPLETE[ExecutionPath ExecutionPath0 completed]

Time cal spent: 30287

Overhead Time: 0

Stage Staget1 spent 1033

Stage Staget2 spent 3088

Stage Staget3 spent 2102

Stage Staget5 spent 356

Stage Staget8 spent 2020

Stage Staget11 spent 5997

Stage Staget14 spent 3681

Stage Staget17 spent 9994

Stage Staget20 spent 287

Stage Staget22 spent 1729

Total is 30287

Test 1 completed

****

TEST 2

****

Initializing test 2

Distance from t1 to t23: 19799.999999999996

Path: [t1, t2, t3, t5, t8, t11, t14, t17, t20, t22, t23]

==========

Service t2 timedout

Alter Service's new estimated time is: 19799.999999999996(PathEstimateTime)+3600(ServiceTimeout)
-1200.0(serviceAverageTime)+1300.0(AltServiceAverageTime)+0(AlgrithmTime)=23499.999999999996

totalPathLimit is 200000

Using service t2 's alter

alg time is :0

Service t17 timedout

Alter Service's new estimated time is: 19799.999999999996(PathEstimateTime)+6600(ServiceTimeout)
-2200.0(serviceAverageTime)+2300.0(AltServiceAverageTime)+0(AlgrithmTime)=26499.999999999996

totalPathLimit is 200000

Using service t17 's alter

alg time is :0

Service t17 timedout

Service t17 's alter also failed, backup looking

look up for t14,

back traced to t14

backuped history nodes [t1, t2, t3, t5, t8, t11]

calculating t14 to end path

New genereated path's etimated time is: 14890(pathExecutedTime)+8700.0(NewPathEstimateTime)
+1(AlgrithmTime)=23591.0

totalPathLimit is 200000

new path created [t1, t2, t3, t5, t8, t11, t14, t18, t20, t22, t23]

alg time is :1

result:

[t1, t2, t3, t5, t8, t11, t14, t18, t20, t22, t23]

COMPLETE[ExecutionPath ExecutionPath1 completed]

Time cal spent: 32244

Overhead Time: 28091

Stage Staget1 spent 239

Stage Staget2 spent 3600

Stage Staget3 spent 1282

Stage Staget5 spent 1060

Stage Staget8 spent 84

Stage Staget11 spent 5489

Stage Staget14 spent 3132

Stage Staget18 spent 1950

Stage Staget20 spent 1395

Stage Staget22 spent 808

Total is 19039

history pathes 0

[t1, t2, t3, t5, t8, t11, t14, t17, t20, t22, t23]

Time cal spent: 28091

Overhead Time: 0

ROLLEDBACK[ExecutionPath ExecutionPath0 rolledback to ExecutionPath ExecutionPath1]

[ExecutionPath ExecutionPath1]

Stage Staget1 spent 239

Stage Staget2 spent 3604

Stage Staget3 spent 1282

Stage Staget5 spent 1060

Stage Staget8 spent 84

Stage Staget11 spent 5489

Stage Staget14 spent 3132

Stage Staget17 spent 13201

Total is 28091

Test 2 completed

BIBLIOGRAPHY

[1] Aho, A. V., Lam, M. S., Sethi, R., and Ullman, J. D. (2006). Compilers: Principles, Techniques, and Tools, 2nd Edition, Addison Wesley.

[2] Ajmone, M., Conte, G., Balbo, G. (1984). A Class of Generalized Stochastic Petri Nets for the Performance Evaluation of Multiprocessor Systems, *ACM Trans. Computer Systems*, *2*(2), 93–122.

[3] Arief, L.B. and Speirs, N.A. (2000). A UML Tool for an Automatic Generation of Simulation Programs, *ACM Proceedings of the Second International Workshop on Software and Performance*, Ottawa, Canada, September 18-20, 71–76, ACM Press.

[4] Balsamo, S. and Marzolla, M. (2005). Performance Evaluation of UML Software Architecture with multiclass queuing network models. *Proceedings of the Fifth International Workshop on Software and Performance*, Palma, Illes Balears, Spain, July 12-14, 37–42, ACM Press.

[5] Baresi, L. and Guinea, S. (2005). Towards Dynamic Monitoring of WS-BPEL Processes, *Proceedings of the Third International Conference on Service-Oriented Computing*, Amsterdam, The Netherlands, December 12-15, 269–282, Springer.

[6] Benatallah, B., Casati, F., and F. Toumani (2006). Representing, Analyzing and Managing Web Service Protocols. *Data and Knowledge Engineering*, *58*(3), 327–357, Elsevier.

[7] Bennett, A. and Field, A. (2004). Performance Engineering with the UML Profile for Schedulability, Performance and Time: A Case Study, *Proceedings of the 12th IEEE International Symposium on Modeling, Analysis, and Simulation of Computer and Telecommunications Systems*, Volendam, The Netherlands, October 5-7, 67–75, IEEE Computer Society Press.

[8] Billington, J., Christensen, S., Hee, K., Kindler, E., Kummer, O., Petricci, L., Post, R., Stehno, C., and M. Weber (2003). *The Petri Net Markup Language: Concepts, Technology, and Tools*, Available at $http://www.informatik.hu-berlin.de/top/pnml/about.html$.

[9] Booth, D., Haas, H., McCabe, F., Newcomer, E., Champion, M., Ferris, C., Orchard, D. (2004). Web Services Architecture, W3C Working Group Note 11. $http://www.w3.org/TR/ws-arch/$.

[10] Bose, R., Helal, A., Sivakumar, V., and Lim, S. (2007). Virtual Sensors for Service Oriented Intelligent Environments. *Proceedings of the Third IASTED International Conference on Advances in Computer Science and Technology*, Phuket, Thailand, April 2-4, 165–170.

[11] Bray, T. et al. (2006). *Extensible Markup Language, Version 1.1.* $http://www.w3.org/TR/2004/REC-xml11-20040204/$.

[12] Broadwell, P. M. (2004). *Response Time as a Performability Metric for Online Services*, Technical Report, No. UCB//CSD-04-1324, University of California Berkeley.

[13] Canevet, C., Gilmore, S., Hillston, J., Kloul, L., and Stevens, P. (2004). Analysing UML 2.0 activity diagrams in the software performance engineering process, *ACM SIGSOFT Software Engineering Notes*, *29*(1), 74–78, ACM Press.

[14] Canfora, G., Penta, M.D., Esposito, R., and Villani, M.L. (2005). QoS-Aware Replanning of Composite Web Services, *Proceedings of IEEE International Conference on Web Services (ICWS 2005)*, Orlando, USA, July 11-15, 121-129, IEEE CS Press.

[15] Capek, J. (2001). *Petri net Simulation of Non-deterministic MAC Layers of Computer Communication Networks*, Ph.D. Thesis, Czech Technical University.

[16] Capek, J. (2003). STPNPlay: A Stochastic Petri-net Modeling and Simulation Tool ($http://dce.felk.cvut.cz/capekj/StpnPlay/index.php$).

[17] Chitrakar, R. (2006). *Formal Modeling and Analysis of Business Process Execution Language Specifications Using Petri News*. Masters Thesis, Univeristy of Illinois at Chicago.

[18] Ciardo G. (1994). Petri nets with marking-dependent arc cardinality: Properties and analysis, *Proceeding of the 15th International Conference on Applications and Theory of Petri Nets*, Zaragoza, Spain, June, Lecture Notes in Computer Science, 815, Springer, 179–198.

[19] Cleland-Huang, J., Chang, C.K., Sethi, G., Javvaji, K., Hu, H., and Xia, J. (2002) Automating Speculative Queries through Event-Based Requirements Traceability, *IEEE Proceedings of the Joint Conference on Requirements Engineering*, Essen, Germany, September 9-13, 289–298, IEEE Computer Society Press.

[20] CORBA 3.0 (2007) *http://www.omg.org/technology/documents/formal/corba_2.htm*.

[21] Cortellessa, V., Gentile, M., and Pizzuti, M. (2004) XPRIT: An XML-Based Tool to Translate UML Diagrams into Execution Graphs and Queueing Networks. *Proceedings of the First International Conference on Quantitative Evaluation of Systems*, Enschede, The Netherlands, September 27-30, 342–343. IEEE Computer Society Press.

[22] Cortellessa, V., Mirandola, R. (2000). Deriving a queueing network based performance model from UML diagrams, *Proceedings of the Second International Workshop on Software and Performance*, Ottawa, Ontario, Canada, September 17-20, 58–70. ACM Press.

[23] Cortellessa, V. and Mirandola, R. (2002) PRIMA-UML: a performance validation incremental methodology on early UML diagrams. *Proceedings of the Third International Workshop on Software and Performance*, Rome, Italy, July 24-26, 302–309. ACM Press.

[24] Cortellessa, V., Marco, A.D., Inverardi, P., Mancinelli, F., and Pelliccione, P. (2005) A Framework for the Integration of Functional and Non-functional Analysis of Software Architectures. *Electronic Notes in Theoretical Computer Science*, *116*, 31–44, Elsevier.

[25] Dan, A., Ludwig, H., and Pacifici, G. (2003). *Web Service Differentiation with Service Level Agreements*, White Paper, IBM Corporation.

[26] Downing T. B. (1998) *Java RMI: Remote Method Invocation*, Wiley Publishing.

[27] Fahland, D. and Reisig, W. (2005). ASM-based Semantics for BPEL: The Negative Control Flow. *Proceedings of the 12th International Workshop on Abstract State Machines*, Paris, France, March 8-11, 131–151.

[28] Farahbod, R., Glasser, U. and Vajihollahi, M. (2004). Specication and Validation of the Business Process Execution Language for Web Services. *Abstract State Machines*, 78–94, Springer-Verlag.

[29] Farahbod, R. (2004). Extending and Rening an Abstract Operational Semantics of the Web Services Architecture for the Business Process Ex- ecution Language. Master thesis, Simon Fraser University, Burnaby B.C. Canada, July.

[30] Ferrara, A. (2004). Web services: A Process Algebra Approach. *Proceedings of Second International Conference on Service Oriented Computing*, New York, NY, November 15-18, 242–251, ACM Press.

[31] Fredman, M.L. and Tarjan, R.E. (1987). Fibonacci heaps and their uses in improved network optimization algorithms, Journal of the ACM, *34*(3), 596–615, ACM Press.

[32] Fu, X., Bultan, T. and Su, J. (2004). Analysis of Interacting BPEL Web Services. *Proceedings of 13th International Conference on World Wide Web*, New York, NY, USA, May 17-20, 621–630, ACM Press.

[33] Fukuzawa, K. and Saeki, M. (2002). Evaluating Software Architectures By Coloured Petri Nets, *Proceedings of the 14th International Conference on Software Engineering and Knowledge Engineering*, Ischia, Italy, July 15-19, 263–270. ACM Press.

[34] Gail, M.H. and Green, S.B. (1976). A generalization of the one-sided two-sample Kolmogorov-Smirnov statistic for evaluating diagnostic tests. Biometrics, September, *32*(3), 561–70, Biometric Society.

[35] Girault, C. and Valk, R. (2003). *Petri Nets for Systems Engineering: A Guide to Modeling, Verification, and Applications*, Springer.

[36] van Glabbeek, R.J. (2001). The Linear Time - Branching Time Spectrum I. The Semantics of Concrete, Sequential Processes. *Handbook of Process Algebra*, 3–99, Elsevier.

[37] Gokhale, S. S. and Lu J. (2006). Performance and Availability Analysis of an E-commerce Site, *Proceedings of IEEE 30th Annual International Computer Software and Applications Conference (COMPSAC 2006)*, Chicago, USA, September 17-21, 495-502, IEEE Computer Society Press.

[38] Gomaa, H. and Menasce, D.A. (2000). Design and Performance Modeling of Component Interconnection Patterns for Distributed Software Architectures, *Proceedings of the Second International Workshop on Software and Performance*, Ottawa, Ontario, Canada, September 17-20, 117–126. ACM Press.

[39] Graham, S., et al. (2004). *Building Web Services with Java: Making Sense of XML, SOAP, WSDL, and UDDI*, Second Edition, Sams Publishing.

[40] Grimes, R. (1997). *Professional Dcom Programming*, Birmingham, Wrox Press.

[41] Gu, G. and Petriu, D.C. (2002). XSLT Transformation From UML Models to LQN Performance Models. *Proceedings of the Third International Workshop on Software and Performance*, Rome, Italy, July 24-26, 227–234, ACM Press.

[42] Gu, G. and Petriu, D.C. (2003). Early Evaluation of Software Performance Based on the UML Performance Profile, *Proceedings of the 2003 conference of the Centre for Advanced Studies on Collaborative research*, Toronto, Ontario, Canada, October 6-9, 66–79, IBM Press.

[43] Gurevich, Y. and Huggins, J.K. (1992). The Semantics of the C Programming Language. *Lecture Notes In Computer Science*, *702*, 274–308, Springer.

[44] Gurevich, Y. (1995). Evolving Algebras 1993: Lipari Guide. *Specification and validation methods*, 9–36, Oxford University Press.

[45] Havlak, P. (1997). Nesting of reducible and irreducible loops. ACM Transaction on Programming Languages and Systems, *19*(4), July, 557–567, ACM Press.

[46] Hamadi, R. and Benatallah, B. (2003). A Petri Net-based Model for Web Service Composition. *Proceedings of the Fourteenth Australasian Database Conference (ADC2003)*, Adelaide, Australia, February, 191–200, Australian Computer Society.

[47] Helal, S. et al (2005). Gator Tech Smart House: A Programmable Pervasive Space. *IEEE Computer*, *38*, 50–60, IEEE Computer Society Press.

[48] Holzmann, G.J. (2003) *The SPIN Model Checker: Primer and Reference Manual*, Addison Wesley.

[49] Hummel, K. A. (2006). Enabling the Computer for the 21st Century to Cope with Real-World Conditions: Towards Fault-Tolerant Ubiquitous computing. *Proceedings of the IEEE International Conference on Pervasive Services*, June 26-29, Lyon, France.

[50] Kurt Jensen, An Introduction to the Theoretical Aspects of Coloured Petri Nets, *Lecture Notes in Computer Science*, vol. *803*; *A Decade of Concurrency, Reflections and Perspectives, REX School/Symposium*, 230–272, Springer-Verlag.

[51] Jensen, K. (1995). *Coloured Petri Nets - Basic Concepts, Analysis Methods and Practical Use*, *2*. London, UK: Springer-Verlag.

[52] Kahkipuro, P. (1999). UML based Performance Modeling framework for object-oriented distributed systems. *Proceedings of Second International Conference on the Unified Modeling Language*, Fort Collins, Colorado, USA, October 28-30, 356–371. Springer-Verlag, London, UK. *Lecture Notes in Computer Science*, *1723*.

[53] Kahkipuro, P. (2001). UML-based performance modeling framework for component-based distributed systems. Performance Engineering, State of the Art and Current Trends, *Lecture Notes in Computer Science*, *2047*, 167–184.

[54] King, P. and Pooley, R. (1999). Using UML to derive stochastic Petri net models. *Proceedings of the 15th UK Performance Engineering Workshop*, Department of Computer Science, the University of Bristol, July 22-23, 45–56, University of Bristol, Bristol.

[55] King, P. and Pooley, R. (2000). Derivation of Petri net Performance Models from UML Specifications of Communications Software, *Proceedings of the 11th International Conference on Computer Performance Evaluation: Modelling Techniques and Tools*, Schaumburg, Illinois, March 27-31, 262–276, Springer-Verlag, London, UK.

[56] King, J., Bose, R., Yang, H.-I., Pickles, S., and Helal, A. (2006). Atlas: A Service-Oriented Sensor Platform. *Proceedings of the First IEEE International Workshop on Practical Issues in Building Sensor Network Applications*, Tampa, Florida, USA, November 14-16, 630-638, IEEE Computer Society Press.

[57] Koizumi, S., Koyama, K. (2007). Workload-aware Business Process Simulation with Statistical Service Analysis and Timed Petri Net, *Proceedings of IEEE International Conference on Web Services (ICWS'07)*, Salt Lake City, USA, July 9-13, 70–77, IEEE Computer Society Press.

[58] Latella, D. and Majzik, I. and Massink, M. (1999). Towards a Formal Operational Semantics of UML Statechart Diagrams. *Proceedings of the Third International Conference on Formal Methods for Open Object-Based Distributed Systems (FMOODS)*, Florence, Italy, February 15-18, 465, Kluwer.

[59] Lazowska, E., Zahorjan, J., Graham, G. and Sevcik, K. (1984). *Quantitative System Performance Computer System Analysis Using Queueing Network Models*, Prentice-Hall, Upper Saddle River, NJ, USA.

[60] Lee, J., Midkiff, S., Padua, DA (1997) Concurrent Static Single Assignment Form and Constant Propagation for Explicitly Parallel Programs, *Proceedings of 10th International Workshop on Languages and Compilers for Parallel Computing*, Minneapolis, Minnesota, USA, August 7-9, 114–130, Springer-Verlag.

[61] Lopez-Grao, J.P., Merseguer, J., and Campos, J. (2004). From UML activity diagrams to Stochastic Petri nets: application to software performance engineering, *Proceedings of the Fourth international workshop on Software and performance*, Redwood Shores, California, USA, January 14-16, 25–36, ACM Press, *ACM SIGSOFT Software Engineering Notes*, *29*(1).

[62] Lohmann, N., Massuthe, P., Stahl, C. and Weinberg, D. (2008). Analyzing Interacting WS-BPEL Processes Using Flexible Model Generation. *Data and Knowledge Engineering*, *64*(1), 38–54, Elsevier.

[63] Ludwig H. (2003). Web Services Qos: External SLAs and Internal Policies. *Proceedings of the Fourth International Conference on Web Information Systems Engineering Workshops*, Rome, Italy, December 13, 115–120, IEEE Computer Society Press.

[64] Magee, J. and Kramer, J. (1999). *Concurrency: State Models and Java Programs*. John Wiley and Sons, Chichester, England.

[65] Marsan, M.A., Balbo G., Conte G. (1986). *Performance Models of Multiprocessor System*s, MIT Press, Cambridge, MA.

[66] Marsan, M.A., Balbo G., Conte G., Donatelli S. and Franceschinis G. (1995). *Modeling with Generalized Stochastic Petri nets*, John Wiley and Sons, West Sussex, England.

[67] Martens, A., Moser, S., Gerhardt, A., Funk, K. (2006) Analyzing compatibility of BPEL processestowards a business process analysis framework in IBMs business integration tools, *International Conference on Internet and Web Applications and Services (ICIW06)*, Guadeloupe, French Caribbean, February 23-25, 147–147, IEEE Computer Society Press.

[68] Martens, A. (2005). Analyzing Web Service Based Business Processes. *The Eighth International Conference on Fundamental Approaches to Software Engineering 2005*, Edinburgh, Scotland, April 4-8. *Lecture Notes in Computer Science*, Vol. *3442*, 19–33, Springer-Verlag.

[69] Marzolla, M. and Mirandola, R. (2007) Performance Prediction of Web Service Workflows. *Proceedings of Third International Conference on the Quality of Software-Architectures (QoSA 2007)*, Medford, Massachusetts, USA, July 12-13, 127–144, Springer.

[70] Marzolla, M. (2004) *Simulation-Based Performance Modeling of UML Software Architectures*, Ph.D. Thesis, Università Ca' Foscari di Venezia.

[71] Marzolla, M. and Balsamo, S. (2004). UML-PSI: the UML Performance Simulator, *Proceedings of the First International Conference on the Quantitive Evaluation of Systems*, Enschede, The Netherlands, September 27-30, 340–341. IEEE Computer Society Press.

[72] McCoy, D.W. and Natis, Y.V. (2003). Service-Oriented Architecture: Mainstream Straight Ahead. *Gartner Research Report, Gartner Inc.*.

[73] Miguel, M.D., Lambolais, T., Hannouz, M., Betge-Brezetz, S., and Piekarec, S. (2000) UML extensions for the specifications and evaluation of latency constraints in architectural models. *Proceedings of the Second International Workshop on Software and Performance*, Ottawa, Ontario, Canada, September 17-20, 83–88. ACM Press.

[74] Milanovic, N. and Malek, M. (2004). Current Solutions for Web Service Composition. *Internet Computing*, *8*, November - December, 51–59, IEEE Computer Society Press.

[75] Moser, S., Martens, A., Gorlach, K., Amme, M., Godlinski, A. (2007). Advanced Verification of Distributed WS-BPEL Business Processes Incorporating CSSA-based Data Flow Analysis, *Proceedings of IEEE International Conference on Services Computing (SCC 2007)*, Salt Lake City, USA, July 7-11, 98–105, IEEE Computer Society Press.

[76] Murata, T. (1989). Petri Nets: Properties, Analysis and Applications, *Proceedings of The IEEE*, *77*(4),541–579.

[77] Murphy, P., Gilpin, M. and Hogan, L. (2006). *Got Legacy? Four Fates Await Your Applications*, Forrester Research.

[78] Murphy, P., Orlov, L. M., Belanger, B. (2006). *Got Legacy? Migration Options For Applications*, Forrester Research.

[79] Nesi, M. (1992). Mechanizing a Proof by Induction of Process Algebra Specifications in Higher Order Logic. *Proceedings of the Third Workshop on Computer Aided Verification 1991, Lecture Notes in Computer Science, 575*, 288–298, Springer.

[80] Nguyen, X. T. Kowalczyk, R. and Phan, M. T. (2006). Modeling and solving qos composition problem using fuzzy disCSP, *Proceedings of IEEE International Conference on Web Services (ICWS'06)*, Chicago, USA, September 18-22, 55–62, IEEE Computer Society.

[81] OASIS (2007). Web Services Business Process Execution Language Version 2.0. *http* : $//docs.oasis - open.org/wsbpel/2.0/OS/wsbpel - v2.0 - OS.html$, April.

[82] Object Management Group (OMG) (2002). *UML profile for schedulability, performance and time specification*. Final Adopted Specification ptc/02-03-02, OMG.

[83] Ouyang, C. et al (2007). Formal semantics and analysis of control flow in BPEL. *Science of Computer Programming*, *67*(2-3), 162–198, Elsevier.

[84] Peterson, J.L. (1977) Petri nets. *ACM Computing Surveys*, *9*(3), 223–252, ACM Press.

[85] Peterson, J.L. (1981). *Petri Net Theory and the Modeling of Systems*, Prentice Hall.

[86] Petriu, D.C. and Shen, H. (2002). Applying the UML Performance Profile: Graph Grammar-Based Derivation of LQN Models from UML Specifications, *Proceedings of the 12th International Conference on Computer Performance Evaluation, Modelling Techniques and Tools*, London, UK, April 14-17, 159–177, Springer-Verlag. *Lecture Notes in Computer Science*, *2324*.

[87] Platform Independent Petri net Editor 2. (2006) $http : //pipe2.sourceforge.net/$.

[88] Pooley, R. (1999). Using UML to derive stochastic process algebras models. *Proceedings of the 15th UK Performance Engineering Workshop*, Department of Computer Science, the University of Bristol, July 22-23, 23–33, University of Bristol, Bristol.

[89] Pooley, R.J. and P. King, J.B. (1999). The Unified Modeling Language and performance engineering. *IEE Proceedings of Software*, *146*(1), 2-10.

[90] Ramalingam, G., (1999) Identifying loops in almost linear time, ACM Transactions on Programming Languages and Systems, *21*(2), 175–188, ACM Press.

[91] Reyes Alamo, J.M., Wong, J. (2008) Service-Oriented Middleware for Smart Home Applications. *Proceedings of IEEE Wireless Hive Networks Conference.*

[92] Rud, D., Schmietendorf, A., and Dumke, R. (2006) Performance Modeling of WS-BPEL Based Web Service Compositions, *Proceedings of IEEE Services Computing Workshops (SCW'06)*, September, Chicago, USA, 140–147, The Institution of Electrical Engineers.

[93] Srinivasan R. (1995) RPC: Remote Procedure Call Protocol Specification Version 2, *http* : $//tools.ietf.org/html/rfc1831$.

[94] Schmidt, K. and Stahl, C. (2004). A Petri net semantic for BPEL4WS - validation and application. *Proceedings of the 11th Workshop on Algorithms and Tools for Petri Nets (AWPN04)*, University of Paderborn, Germany, September 30 - October 1, 1–6.

[95] Smart Home Research Lab (2005), Iowa State University, $http://smarthome.cs.iastate.edu/$.

[96] Smith, C.U. and Williams, L.G. (2002). *Performance Solutions: A Practical Guide to Creating Responsive, Scalable Software*, Addison Wesley, Boston, MA, USA.

[97] Gudgin, M., Hadley, M., Mendelsohn, N., Moreau, J.J., Nielsen, H.F., Karmarkar, A., Lafon, Y. (2007) Simple Object Access Protocol (SOAP) version 1.2, $http://www.w3.org/TR/soap/$.

[98] Sreedhar, V. C., Gao, G. R., and Lee, Y.-F. (1996). Identifying loops using DJ graphs. ACM Programming Languages and Systems, *18*(6), November, 649–658, ACM Press.

[99] Srivastava, B. and Koehler, J. (2003). Web Service Composition - Current Solutions and Open Problems. *Proceedings of the International Conference on Automated Planning and Scheduling*, Trento, Italy, June 9-13, 28–35.

[100] Standish group (1994) *The Chaos Report*, survey report.

[101] Tarjan, R. E. (1983). Data Structures and Network Algorithms. Society for Industrial and Applied Mathematics, Philadelphia, PA.

[102] Bellwood, T., et al (2004) Universal Description, Discovery and Integration (UDDI) version 3.0.2, *http://uddi.org/pubs/uddi_v3.htm.*

[103] Verbeek, H.M.W. (1997). Verification of workflow nets. *Proceedings of 18th International Conference on Application and Theory of Petri Nets*, Toulouse, France, June 23-27, 407–426. Lecture Notes in Computer Science, *1248*, Springer-Verlag.

[104] Verbeek, H.M.W. and van der Aalst, W.M.P. (2005). Analyzing BPEL processes using Petri nets. *Proceedings of the Second International Workshop on Applications of Petri Nets to Coordination, Workflow and Business Process Management*, Miami, USA, June 20, 59–78.

[105] Weyuker, E.J. and Vokolos, F.I. (2000). Experience with Performance Testing of Software System: Issues, an Approach, and Case Study. *IEEE Transaction Software Engineering*, *26*(12), 1147-1156, IEEE Press.

[106] Christensen, E., Curbera, F., Meredith, G., Weerawarana, S. (2001) Web Service Description Language (WSDL) 1.1, $http://www.w3.org/TR/wsdl$.

[107] Xia, J. and Chang, C. K. and Wise, J. and Ge, Y.(2006). An Empirical Performance Study: PSIM. *Computer Journal*, *49*(5), 509–526, Oxford University Press.

[108] J. Xia, Y. Ge, and C.K. Chang (2005). An Empirical Performance Study for Validating a Performance Analysis Approach: PSIM, *Proceedings of IEEE 29th Annual International Computer Software and Applications Conference (COMPSAC 2005)*, Edinburgh, Scotland, UK, July 26-28, 307–312, IEEE Computer Society Press.

[109] Xia, J. (2006). QoS-based Service Composition. *Proceedings of IEEE 30th Annual International Computer Software and Applications Conference (COMPSAC 2006), student paper*, September 17-21, 359–361, IEEE Computer Society.

[110] Xia, J., Chang, C.K., Liu, B., Jaygarl, H., Lu, K.S. (2008). WS-Pro: the Performance-driven Service Composition Framework, Invited journal paper, submitted to *Journal of Systems and Software*, Elsevier.

[111] Xia, J. and Chang, C.K. (2006). Performance-driven Service Selection Using Stochastic CPN. *Proceedings of IEEE 2006 John Vincent Atanasoff International Symposium on Modern Computing (JVA 2006)*, October 3-6, 99–104, IEEE Computer Society.

[112] Xia, J., Chang, C.K., Kim, T.H., Yang, H.I., Bose, R. and Helal, S. (2007). Fault-resilient Ubiquitous Service Composition, *Proceedings of the third IET International Conference on Intelligent Environments (IE'07)*, Ulm, Germany, September 24-25, 108-115.

[113] Yamato, Y., Tanaka, Y., and Sunaga, H. (2006). Context-aware ubiquitous service composition technology. *Proceedings of the IFIP International Conference on Research and Practical Issues of Enterprise Information Systems*, Vienna, Austria, April 24-26, 51–61, Kluwer.

[114] Yang, H.I., Bose, R. and Helal, S., Xia, J., Chang, C.K., Kim, T.H. (2008). Fault-resilient Ubiquitous Service Composition, Book Chapter in *Advanced Intelligent Environments*, H. Hagrass, Editor, Springer Verlag.

[115] Yu, T. and Lin, K. J. (2004). The design of qos broker algorithms for qos-capable web services, *International Journal of Web Services Research*, *1*(4), 33–50, Igi Publishing.

[116] Zeng, L., Benatallah, B. and Dumas, M., (2003) Quality Driven Web Services Composition, *Proceedings of 12th International Conference on World Wide Web (WWW 2003)*, Budapest, Hungary, May 20-24, 411-421, ACM Press.

[117] Zeng, L., Benatallah, B., Ngu, A. H.H., Dumas, M., Kalagnanam, J., Chang, H. (2004) QoS-aware middleware for web services composition, *IEEE Transactions on Software Engineering*, *30*(5), 311-327, IEEE Press.